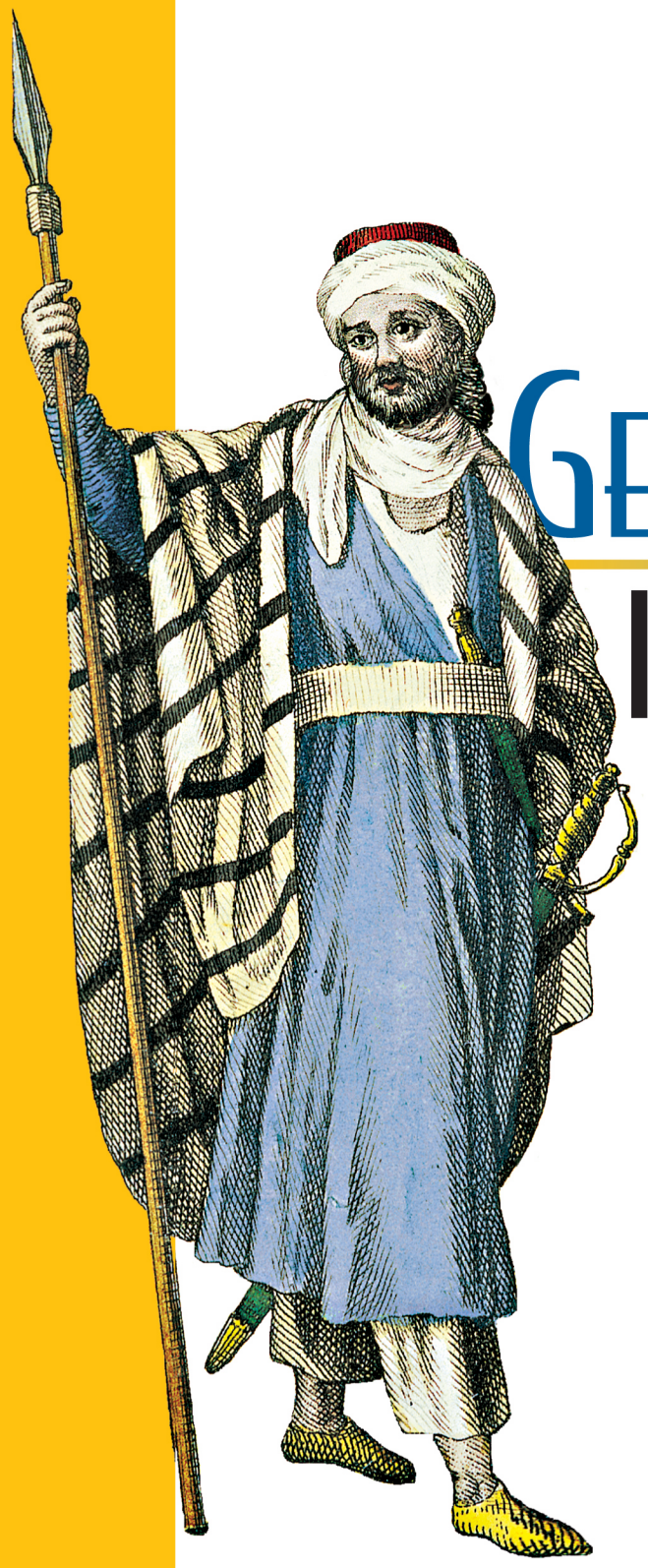

Code Generation in Action

Jack Herrington

MANNING

brief contents

CHAPTER 1

Overview

Code generation is about writing programs that write programs. With today's complex code-intensive frameworks, such as Java 2 Enterprise Edition (J2EE), Microsoft's .NET, and Microsoft Foundation Classes (MFC), it's becoming increasingly important that we use our skills to build programs which aid us in building our applications. Generally speaking, the more complex your framework, the more appealing you will find a code generation solution.

This book is the first to cover the breadth of high-level code generation at the theory level. As we drill down into implementation, we show you how to create generators ranging from very simple coding helpers that you can write in an hour or two, to complex generators that build and manage large portions of your application from abstract models. The book will also aid you in using generators which you can find on the shelf—we explain not only how to find the generators appropriate to your task, but also how to customize and deploy them.

To understand why generators are so valuable, let's start with a hypothetical case study illustrating the building of applications using generation.

1.1 A GENERATION CASE STUDY

In this section, we examine a case study that shows the potential for code generation in modern application development. Our case study examines a corporate accounting application. Because of the complex nature of the accounting work, the initial schema has 150 database tables. By any estimation, it is a massive effort. The requirements specify that the application work in both a client/server mode as well as over HTTP using a web interface.

We chose Java and Enterprise JavaBeans (EJB) as our back-end database access layer, JavaServer Pages (JSP) for the web pages, and Swing for the client portion of the client/server model. This is a standard J2EE architecture. Figure 1.1 shows the block diagram for the accounting application.

The design is simple and standard. The web client comes in through the Tomcat/JSP layer, and the desktop client comes in through a Swing interface that will talk directly (over Remote Method Invocation, or RMI) to the database access layer. This common database access layer encapsulates both database persistence and business logic. The database access layer, in turn, talks directly to the database.

1.1.1 Step 1: generating the database access layer

The code generation portion of the story starts with the building of the database access layer. We establish two teams: one for the database access layer and one for the user interface. We'll start with the database access team.

Our implementation of the EJB architecture specifies five classes and two interfaces per table. This is not a usual object/relational mapping in the EJB model. Each class or interface is in its own file. These seven files make up a single EJB "entity." Seven files multiplied by 150 tables tells us that we are already looking at 1,050 files for the EJB entities. It's a big number, but it is not unusual for a J2EE system with this many tables.

We build from scratch four EJB entities all the way to completion. These entities provide a cross section of the various table structures we have in our schema. Building just these files takes four man-weeks.

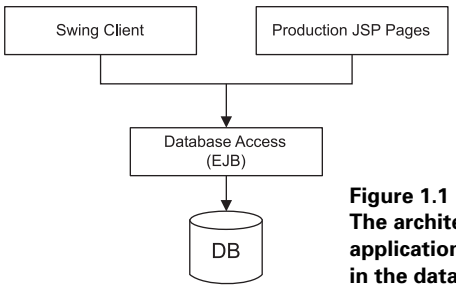

Figure 1.1
The architecture of a J2EE accounting application. The business logic is included in the database access layer.

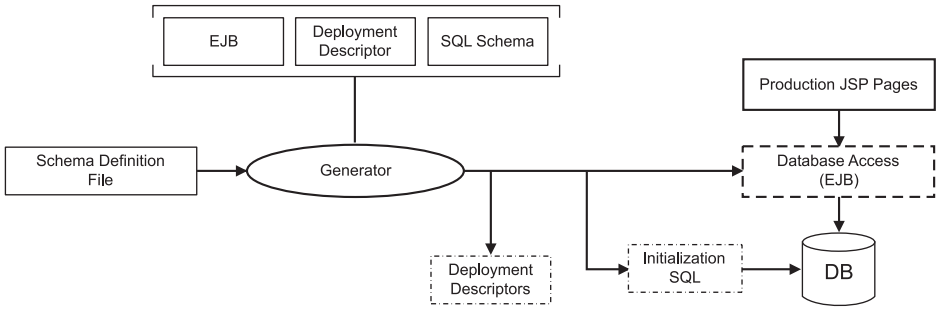

Figure 1.2 The database generator builds the database access classes.

At this point, the schedule looks bad, with only four tables built in four weeks. By extrapolating this timetable, we calculate that the database access layer alone will take three man-years of effort. We must reduce that schedule significantly.

The first thing we notice about the EJB code is that it is pretty standardized, which means it is going to entail a lot of monotonous work. We've had some experience with generators in the past and think that a code generator can be successful in this case. Our previous experience tells us that the first version of the generator will take about two weeks to build. We decide to take the risk and spend two weeks building a generator to create the EJBs.

Using the four example EJBs we have already built, we create templates for a template-based generator. As you'd expect, a template-based generator uses a page-template language (e.g., JSP, ASP, or PHP) to build code. Instead of creating web pages dynamically, we build class implementation files or Extensible Markup Language (XML) deployment descriptors. Figure 1.2 shows the processing flow of our database access layer generator.

The generator takes an XML file as input. This file defines the tables and fields of the database and feeds this input to a series of templates—one template per output file type—and stores the output of the template in the target files. These target files include the database access EJBs (all seven files per table), the Structured Query Language (SQL) for the database, and the deployment descriptors.

This technique is known as *model-driven generation*. The model of the system is abstracted into the schema definition file, which is used to build large portions of the production system. The model can be generated by hand or by using a tool such as Rational Rose, which can export Unified Modeling Language (UML) in XML form.

Our first pass of the generator builds the EJB files, the deployment descriptors, and the schema file for the database. Rather than start with all 150 tables, we pick 10 that together provide a solid test set of the various table types.

The initial results are encouraging. The SQL for the first 10 database tables is built properly on the first try. The EJB classes have some initial problems, but we are able

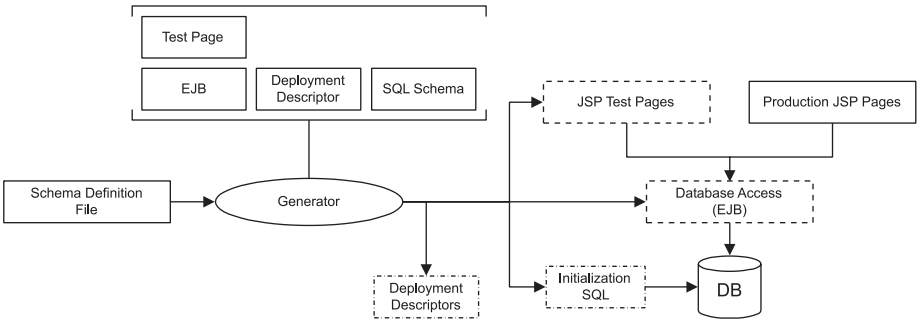

Figure 1.3 The generator builds the JSP test pages as well as all the previous output files.

to debug the classes and then alter the templates accordingly. Running the generator fixes bug occurrences across all the EJBs.

This is a good start, but we want an end-to-end test—one that will show the system working through a web server. We decide to upgrade the generator to have it also build a set of simple JSP test pages that will work through the EJB layer. This will not be the finished interface, but it will allow us to create a simple unit test of our database access layer. At this point the generator block diagram looks like the one shown in figure 1.3.

Again the results are encouraging. The generator builds test pages that will test the EJB session beans to make sure the whole system works from end to end. To make it simple, our test pages map one to one with the session beans. We know that there will be a more complex mapping between entities and production pages because of the requirements of the user interface.

Now we are able to make some observations about the code generation workflow and its benefits:

- Our work on the generator itself is both interesting and motivating. It also broadens our engineering experience.
- We are moving much more quickly toward completion than we would have if we had been hand-coding.
- We are pushing our understanding of both EJB and JSP and finding issues early that we believe we would not otherwise have found until much later.
- We experiment with various types of EJB implementation styles to spot performance or deployment issues. Having the generator means we can write the code one way and try it, and then change the template to try another approach.

Code generation for the database access layer is no longer an option; it is now the implementation model. The next step is to complete the schema definition XML file, which we accomplish with the aid of an XML editing tool such as Altova's XMLSpy.

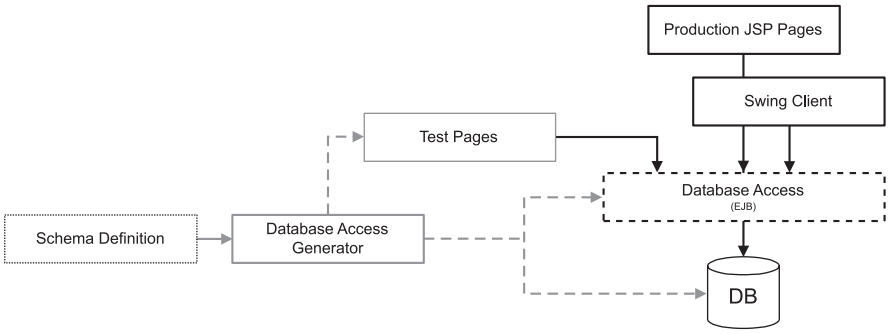

Figure 1.4 How the database generator fits into the accounting application architecture

Figure 1.4 shows how the database generator builds portions of the complete system. In the figure, the production units have a solid black border and the generator has a dark-gray border. A light-gray border indicates an output file, and user-editable files have a dotted light-gray border.

An accounting application is not just schema; it also includes business logic that defines how the schema is used. The schema definition file shown in figure 1.4 is a combination of files that includes the schema and extensions for custom business logic. These are merged together during the construction of the database access classes. We find that putting the business logic and the schema together in a single place makes maintenance and synchronization much easier.

In total, the generator for the database access layer takes four weeks to design, implement, and test—a little longer than the anticipated two weeks, but we can deal with that given the productivity gains. (This timeframe is somewhat misleading because the generator and its templates are software and, as such, require long-term maintenance as the project evolves.)

Some initial project metrics

Our estimation of how long it would have taken to hand-code all 1,050 classes is three man-years; it takes us one man-month to build the generator. This represents a significant reduction in our schedule. The resulting EJBs are much more consistent than any we would have handwritten, and they contain all of the functionality we could want. In addition, if large-scale changes are required we can alter the templates and roll them out across the entire code base in seconds.

It's not all about schedule time, though; there are other metrics to analyze. Table 1.1 compares hand-coding and code generation for the database access layer.

Table 1.1 A comparison of hand-coding and code generation for the database access layer

Hand-Coding	Code Generation
Each new class slightly increases on the quality of the previous one. There is a lot of copy-and-pasted code. The code base is of inconsistent quality across the entities.	The code quality is consistent across all of the entities.
Making additions means altering every entity one by one.	When mass changes are required, the templates are updated with the new code and the generator is rerun.
Bugs are fixed one by one across the entities. When a bug affects more than one class, each must be hand-fixed and verified. Some changes made in base classes do not suffer from this problem.	Bugs in the classes are fixed by changes to the templates. When the generator is rerun, all of the classes get the fixes automatically.
Each class has a corresponding unit test that fits within the unit test framework.	Unit tests are critical. We use text differencing to make sure that the generator is still creating the proper implementations. We compare the latest generation against the known goods. In addition we can author or generate classical unit tests as we would by hand-coding.
A compatibility layer is required underneath the code to move it to a different framework.	Because the schema and business logic are stored in a meta description, the generator and templates can be altered to build code for a different language or framework.
The maintenance of the schema is a separate task from the maintenance of the corresponding EJB entity. It is likely that the field definitions in the database will diverge from the corresponding variable definitions in the EJB entities and cause aberrant behavior.	The schema and the entity classes are created at the same time by the same mechanism. Synchronization is automatic. If there is a synchronization problem it is the fault of the generator, so the issue can be addressed easily by fixing and rerunning the generator.

1.1.2 Step 2: generating the user interface

While the database access layer team is busy building the generator, the front-end team is hard at work designing the user interface for our application.

Once the database access team has finished with the first four EJB entities, the user interface team develops all of the necessary pages to support them by hand. For each table, it takes one man-week.

After this prototype phase, we take account of where we are. We know that the engineers will improve their speed over time, but with padding, the schedule for the web user interface is projected to take three man-years to cover all 150 tables with JSP pages.

We decide to use generation for the user interface as well. Given our success with a template-based generator for the database access layer, we decide to use a similar

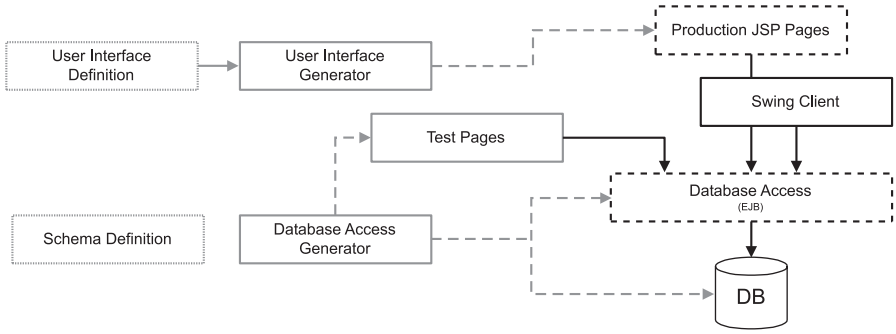

Figure 1.5 The user interface generator building the production JSPs as shown in relation to the database generator

technique for the user interface. We use the pages we built to cover the four database access EJB entities as the basis of the templates. Then we build a user interface generator that takes an XML definition file containing all of the information about the beans and use the templates to generate JSP pages. Figure 1.5 adds the user interface generator to our diagram.

The generator takes six man-weeks to design, build, and test. Even with the first version complete, we make sure that everyone understands that the user interface generator is an ongoing project that will require constant maintenance.

The user interface definition is actually a combination of raw field data and custom options that can be set on a page-by-page basis. That allows for some customizations to be made to the interface on a case-by-case basis.

We hand-code the user interface definition for our four prototype EJBs and use the generator to build the pages. It takes four seconds to build the pages.

That's great, but we still have to create the definitions for all 150 EJB entities. That would be a pain. Instead, the front-end team works with the back-end team to alter the database access generator so that it builds the entity definitions for the user interface.

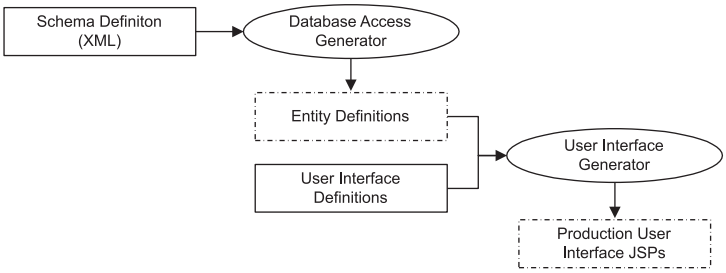

Figure 1.6 The database generator building entity definitions, which are used by the user interface generator in the process of building the JSPs for the production interface

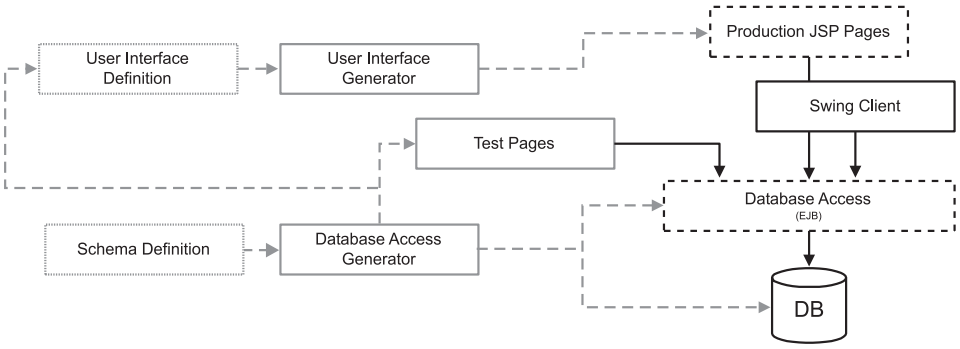

Figure 1.7 The user interface generator now gets part of its information from the database generator. All of the other generator components remain the same.

The two generators now cascade. The output of the database generator is fed into the user interface generator that created the production web interface. Together the two generators build all 150 entities and page sets in four minutes. Figure 1.6 illustrates this cascading architecture. Figure 1.7 shows the linkage between the database access generator and the user interface generator.

Once again, we've turned several man-years of effort into a couple of man-months.

1.1.3 Step 3: building the client interface

Our success with the database and user interface generators make it an easy decision to build a Swing generator. Our experience with building the user interface generator helps us when we work with the graphic designer to simplify the interface to make it easy to generate.

To build the Swing generator, we alter the interface generator to build both JSP and Swing interface code. Developing the templates and altering the generator takes another four weeks. Figure 1.8 shows the addition of the Swing client outputs to the user interface generator.

Our original estimate for building Swing for each entity was two to three days, which meant one and a half man-years for the entire process. Our four weeks of development to support Swing using the generator is a big improvement over that estimate.

These schedule impacts sound almost too good to be true, and in a sense, they are. When you evaluate the productivity benefit of a generator, you should always attempt to compare the schedule of the direct coding tasks against the time taken to develop, deploy, and maintain the generator. You will still need to spend time debugging, building tests, and building custom code that can't be generated.

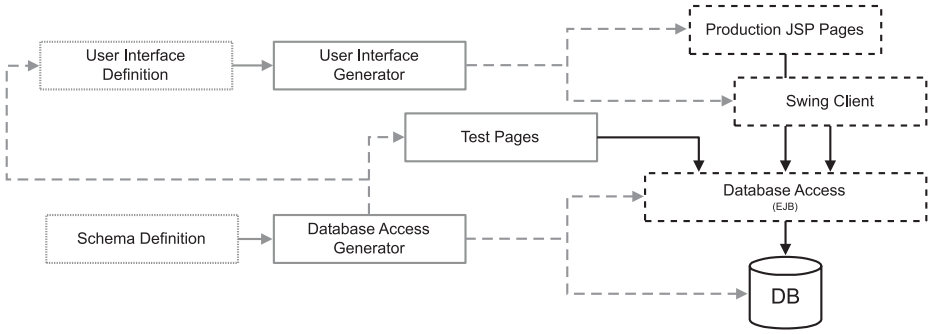

Figure 1.8 The user interface generator builds both the Swing and the JSP code.

1.1.4 Step 4: building unit tests

At this point, the power that the generators hold over our project is a bit intimidating—one small glitch in the templates could invalidate almost every class or page in the system. We need unit tests to ensure that the system runs as expected after the generators are run.

First we need a test data set. The QA group uses an automated tool to preload the data set from the user interface. Now that the database is populated, we can extract the data from the database into a set of test data files. We can then use a generator to build a data loader that will load the test data files through the EJB layer and check the output. This becomes our first unit test tool. Figure 1.9 shows our system with the addition of the unit test generator.

Now whenever the generator is run, we can run a unit-test pass on the output to ensure that the interfaces perform properly.

One enterprising engineer takes it a step further. Because the JSPs built by the generator are so consistent, he is able to build a *user agent* robot using Perl and `LWP::Simple` that will automatically log into the site, virtually walk around the site

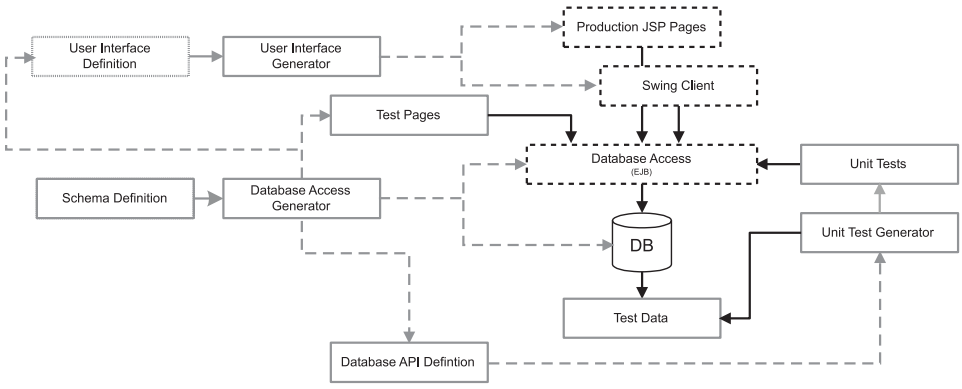

Figure 1.9 The addition of the unit test generator, which tests the database access layer

as if it were a user, and load the test data through the JSP interface. This provides a convenient system test that will check the entire technology stack from end to end. Figure 1.10 shows how the user agent generator connects through the user interface into the system.

1.1.5 Step 5: integrating a technical change

After two weeks of testing, we discover a bug that involves the storage and display of very large numeric values. After some direct debugging against the generated code, we find that the fix requires a change to the definition of 90 percent of the fields in the database, as well as changes to the corresponding type definitions in the majority of the Java classes.

Altering the templates within the generator takes a day. Running the generators requires four minutes. Unit and manual testing consumes two more days.

We calculate that manually making these fixes across all of the Java classes and across the database schema definition would take six months of work and that the code we would have at the end of the process would be of questionable quality.

As you can guess, we are pleased with the difference in outcomes between code generation and hand-coding when it comes to altering the entire code base.

1.1.6 Step 6: integrating a design change

Using code generators has accelerated development to the point where we are able to show some reasonably useful demos early in the process. The demos give both the engineering team and the product marketing group insight into the shortcomings with the original application specification. It is obvious that extra functionality is necessary, to the tune of about 20 new tables. Substantial changes are also required in the structure of some of the existing tables.

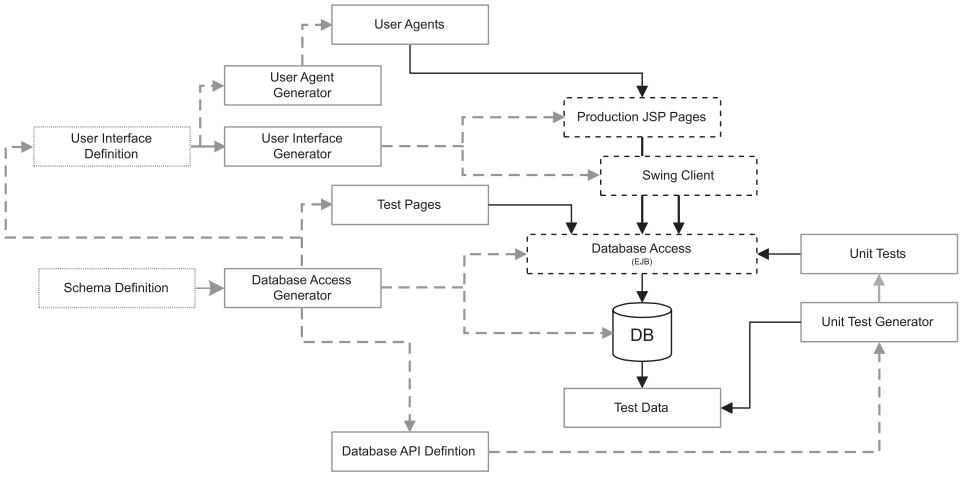

Figure 1.10 The user agent generator builds robots that test the application using the JSP interface.

We are able to react to these changing business requirements by altering our schema definition files and regenerating. After a few iterations and some alterations to the templates, we are able to demo the rudiments of the new schema and refine the feature set even further.

It's almost impossible to calculate how far we would have progressed toward project completion if we had been hand-coding. Discovering the flaws in the design would have necessitated making a serious business decision: either rewrite major portions or maintain the faults in the design. Using code generation, we are able to quickly react to customer demands and make large-scale changes to the structure of the software.

1.1.7 Step 7: building an RPC layer

One of the items on the "wish list" for the project is a Remote Procedure Call (RPC) layer. This had originally fallen off the list because of schedule pressures. Now that we are ahead of schedule, we decide on the fly to take the next step of building a Simple Object Access Protocol (SOAP) layer on top of our application. The RPC layer will go on top of our database access layer. Using the API definitions file that is already being generated for the user interface generator, we create another RPC generator that builds not only the server side of the SOAP layer but also client adapters for Visual Basic and C++, which makes it easy for our customers to use our APIs.

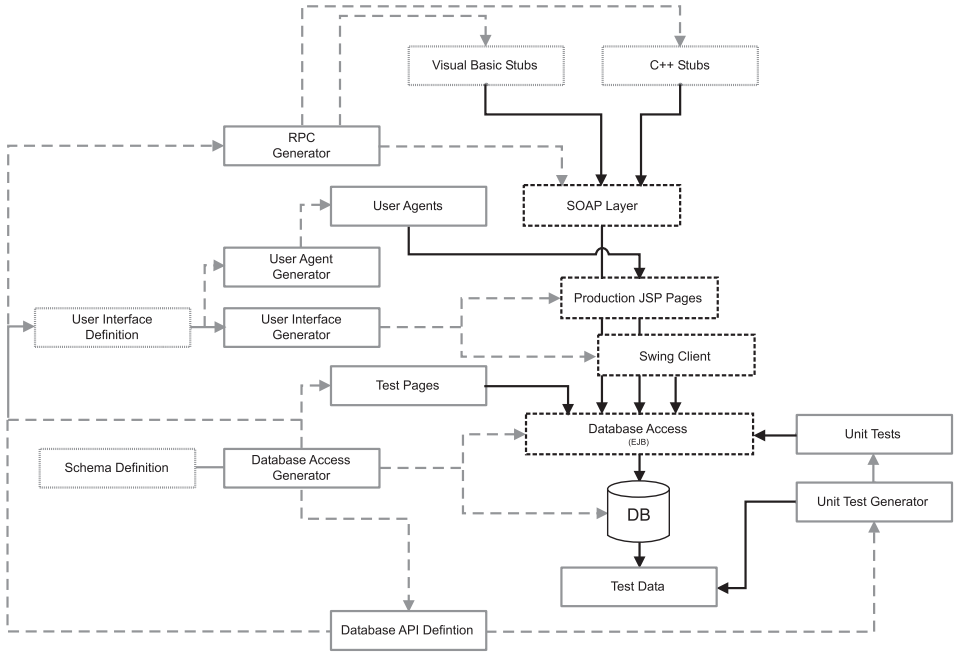

Figure 1.11 We've added an RPC generator that builds not only the server code but the client stub code for both Visual Basic and C++.

Figure 1.11 shows the entire RPC layer. This includes the RPC generator, which in turn builds the SOAP layer, the Visual Basic code, and the C++ stubs.

To test the RPC layer, we use a derivative of the user agent test system. This new robot uses Perl and `SOAP::Lite` to talk to the SOAP layer to implement the test data-loading.

1.1.8 Step 8: building documentation

The final engineering task before deployment is to build documentation both for the RPC layer as an external resource and for the internal APIs as a programmer reference. Using a documentation generator, we build RPC documentation using a definition file that is output as part of the RPC layer generation. The internal API documentation is built using standard JavaDoc tools. Figure 1.12 shows our completed system (the JavaDoc output is implied in this figure).

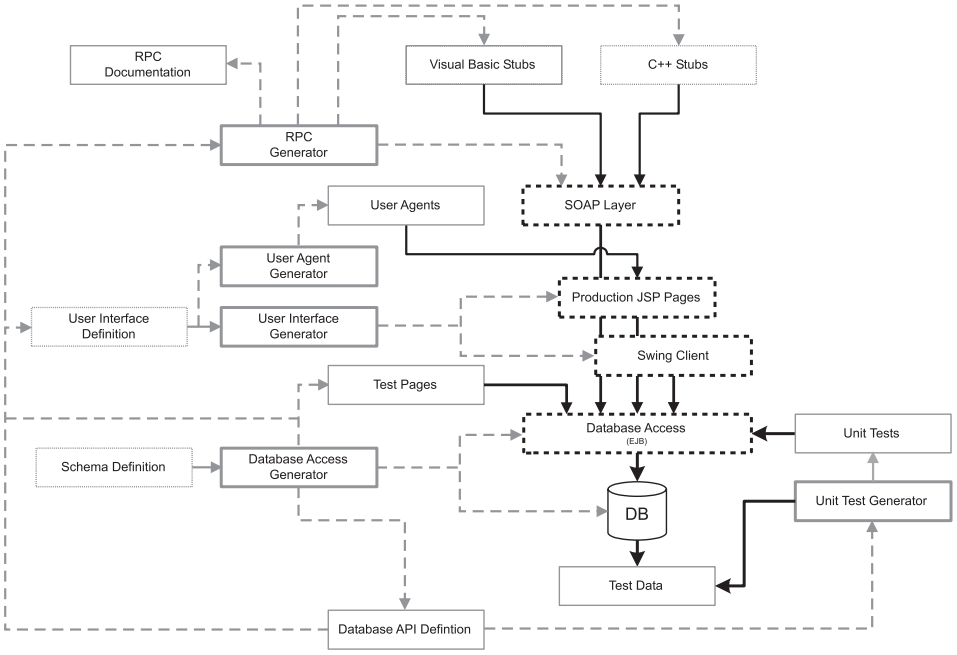

Figure 1.12 The documentation generated for the RPC layer by the RPC generator. The JavaDoc output for the other layers is not shown here.

1.1.9 Case study conclusions

We can make several assertions from this hypothetical case study:

- Code generation has a dramatic impact on development time and engineering productivity.
- The application is amenable to change on a large scale.
- The business rules are abstracted into files that are free of language or framework details that would hinder portability.
- The code for the application is of consistently high quality across the code base.

In the chapters that follow, we describe the techniques we used in this case study in detail.

The case study uses a lot of generation. You may find that your application doesn't need to use that much. Not to worry—this book covers both small and large code generation implementations.

1.2 BENEFITS OF CODE GENERATION FOR ENGINEERS

As the case study shows, code generation techniques provide substantial benefits to software engineers at all levels. These benefits include:

- *Quality*—Large volumes of handwritten code tend to have inconsistent quality because engineers find newer or better approaches as they work. Code generation from templates creates a consistent code base instantly, and when the templates are changed and the generator is run, the bug fixes or coding improvements are applied consistently throughout the code base.
- *Consistency*—The code that is built by a code generator is consistent in the design of the APIs and the use of variable naming. This results in a no-surprises interface that is easy to understand and use.
- *A single point of knowledge*—Using the case study as an example, a change in the schema file percolates through all of the cascading generators to implement the change across the system. Even in the best hand-coded systems, a table name change would involve individual manual changes to the physical schema, the object layer and its documentation, and the test bed. A code generation architecture allows you to change a table name in a single location and then regenerate the schema, the object layer, the documentation, and the test bed to match the new naming requirement.
- *More design time*—The schedule for a code generation project is significantly different than for a hand-coded project. In the schedule for hand-coding large sections, little room exists for analyzing the best use of the system and the APIs. When faulty assumptions are made about use of the framework APIs, then either those decisions are retained or large sections of code have to be

rewritten to use the API appropriately. With code generation, engineers can rewrite the templates to modify how APIs are used and then run the generator to produce the fixed code.

In addition, because code generation compresses time on certain projects, more time can be spent doing adequate design and prototype testing to avoid downstream rework.

- *Design decisions that stand out*—High-level business rules are lost in the minutiae of implementation code. Code generators use abstract definition files to specify the design of the code to be generated. These files are much shorter and more specific than the resulting code. Small exceptions stand out much more clearly in a five-line definition file than in the resulting five hundred lines of implementation code.

To sum up, when you can work *smarter* rather than *harder* and use the computer to offload some of your work, your project will be better off.

1.3 BENEFITS OF CODE GENERATION FOR MANAGERS

Many of the advantages to the engineer should be important to engineering management, such as increased productivity and quality. There are certain aspects, however, that are uniquely important at the business level. Let's take a look at these business-level advantages:

- *Architectural consistency*—The code generator used for a project is the realization of the architecture decisions made upfront in the development cycle. This has three advantages:
  - The generator encourages programmers to work within the architecture.
  - When it is difficult to "get the generator to do what I want it to do," it is a good indication that the new feature does not work within the existing architecture.
  - A well-documented and -maintained code generator provides a consistent structure and approach, even as team members leave the project.
- *Abstraction*—The architectures of the code generators presented in this book have the application logic (business logic, database schema definition, user interface definition, etc.) in language-independent definition files. This abstraction of the semantics of the application from the code that implements the semantics has profound benefits:
  - Engineers will be able to build new templates that translate the logic into other languages, or onto other platforms, much more easily than the equivalent port of handwritten code.
  - Business analysts can review and validate the design in the abstract.

- Capturing the application semantics at the abstract level can aid in the development of work products outside implementation code. These can include various forms of documentation, test cases, product support materials, and so forth.

- *High morale*—Long projects can be tough on teams, and long projects with large amounts of tedious coding can be even worse. Code generation reduces project schedules and keeps the engineers focused on the interesting, unique work, as opposed to grinding through large volumes of tedious code. In addition, because the quality of generated code is uniformly high, the engineering team will have confidence and pride in the code base.
- *Agile development*—A key feature of generated code bases is their malleability. We discussed this topic at a technical level earlier; at the business level, this means that the software will be easier to change and upgrade over the long run.

1.4 THE CODE GENERATION PROCESS

In upcoming chapters, we discuss the development lifecycle and how to build a generator. In this section, we provide a roadmap for the development and deployment of a generator within the engineering organization.

The sections that follow outline the entire lifecycle of the generator—from assessing the need, to development, to maintenance.

1.4.1 Assessing the need

The first step in getting help is admitting you have a problem. So it is with code generation. You first need to see the problem you have at hand and then decide if code generation is the right solution to that problem.

In the case study, the first clue that we needed a code generation solution was the schedule impact of writing 1,050 Java files. The second clue was the monotony of the task of writing the Java code. Both of these are strong indicators for using code generation. If you need a lot of Java files, and writing them is monotonous work, you will want to generate those files.

Once you have assessed the need, you must decide for yourself how you want to present the case to engineering and management. You can choose from three basic strategies: the formal method, the Skunkworks method, and the "my own tool" method.

The formal method

This method involves turning the generator into a full-fledged project from the beginning and following whatever process your company has for the design and implementation of software engineering projects.

The first phase of software design is the requirements phase, in which you gather information about what the generator needs to do. In this phase, you want to agree

on the scope of the work handled by the generator. In particular, you must clarify these issues:

- How much will the generator do in the first release? The scope should be very clear on this issue.
- How much will the generator do in the subsequent releases? This will help define the long-term architecture of the generator.
- For what is the generator responsible? Having clear lines of responsibility will allow you to ensure that key features are implemented and that extraneous features are left out.
- For what is the generator not responsible? Going through the exercise of cataloging what a piece of software is and is not responsible for is very valuable.
- From an outside perspective, is there anything unusual about what it covers or does not cover? Make sure you catalog the areas of responsibility that could be considered unusual to someone just coming into the project.

In addition to clarifying the scope of the project, make sure you establish some basic goals about the development process for the generator. Here are some points you may want to clarify:

- How will the generator fit into the development cycle?
- Will the generator go to the customer or stay in-house?
- What style of generation will it use? It's important to get a consensus on what type of generator to build. This book will present many types of generators as we go along.
- How and when will an engineer use the generator?
- Is it a permanent or temporary solution?
- Who will maintain the generator?
- Who is the arbiter of features for the generator?
- Is the first release of the generator prototype code or production code?
- What development tools will be used?

These types of questions have to be answered in addition to the standard feature specifications.

The advantage of the formal method is that if the project goes forward, it has everyone's blessing. The disadvantage is that the generator project may fail through bad project management, or by having too many features, or by having too little clarity around the feature set.

The Skunkworks method

The term Skunkworks is used when an engineer goes off on her own time and implements a solution and then returns with it to the team. At this point the question of deployment and maintenance becomes an issue for the team to resolve.

The value of this technique is the elimination of all external influences from the process. You think up the project and then build it. If the generator project fails, then nobody knows but you.

Of course, the disadvantage is that you can frighten or alienate other engineers or managers, and you may lose your work if they decide not to use your generator.

The "my own tool" method

The directive comes down on high that all your methods must now have X by next week to conform to the new architectural standard. What do you do? You can either get typing, or you can make a tool to do the work for you. This is the type of scenario that breeds the "my own tool" generator.

The my-own-tool method is the same as the Skunkworks model except that, at the end, the tool remains yours. You don't present it to the team, because they might say it shouldn't be used. As long as the tool is yours, you can use it as you wish, and nobody will be the wiser.

You may find that you have a perception problem if people find out. It's not good being "that nut with that tool nobody understands."

That being said, there is nothing stopping the my-own-tool generator from becoming a Skunkworks generator or being promoted to a formal method generator. All you need to do is tell people about it and do a little educating.

1.4.2 Laying down the infrastructure

After figuring out what problem you are solving and how you are going to solve it, the next step is to get your tools ready. This process involves several steps. The most important is to control and track versioning across the development team. You will also need to set up a test environment, select the full suite of tools that your team will use, and communicate those choices clearly.

Ensure source-code control

Source-code control is so important that I will mention it many times. Source-code control is vital when working with code generators because large blocks of code are rewritten in the blink of an eye.

Make sure that you are familiar with the command-line interface of your source-code control system. Also, be familiar with any check-in or checkout automation facility the system may have. Using the automation facility may be an easy way to integrate your generator into the development workflow. For example, you could have your

documentation generator automatically build documentation upon source code check-in.

You should also check the Internet to see if a wrapper is available for your source-code control system. Perforce, for example, has Perl modules on CPAN that give you access to all of the Perforce functionality through convenient Perl functions.

Build your sandbox

You should use your source-code control system to build a sandbox for the generator and the code it creates—particularly when integrating a generator into an existing system to replace existing code. This allows you to run and test the generator in isolation without worrying about corrupting the main branch and interrupting anyone else's work.

Standardize your development tools

If you are using different development tools for the generator than you are for the application, you will want to spend a little time preparing those tools for deployment. In particular, you should take the time to create an installation kit with just the executables and extra modules required to support the generator. Be sure to document the installation procedure; nothing kills the enthusiasm of an engineering tool deployment quite like installation problems.

This also means locking in the version of the development tool you use. You should look at the versioning of these tools with the same skepticism that you would an update to your C, C++, or Java compiler. Each compiler, interpreter, or plug-in module you use should be frozen at the deployment version and then upgraded with care.

Buy it or build it

At this point, you've laid the groundwork for the development. Now you need to decide whether to buy or build the generator.

Buying a generator is not the end of the game; it is just the beginning. First, you must test and customize the generator against the code you would like to build. Remember, the generator is just a tool. Because you must assume complete responsibility for the code that it generates in production, you should feel completely confident about the behavior of the generator and the code that it creates. Also, standardize the installation of the base package and any customizations that you have made—just as you would with any development tool you use.

The emphasis of this book is on building generators. This approach provides you with full control and also allows you to understand what an off-the-shelf generator is doing for you.

1.4.3 Documenting it

Once the generator is finished with its first release, you should concentrate on documentation. You have to create this documentation regardless of whether you buy or build the generator.

Two basic forms of documentation go along with the generator. The first is the architectural document and will be used by the maintainers of the generator. This document should include:

- The design goal of the generator
- Pros and cons of the generator approach as it applies to the problem at hand
- The block architecture diagram for the generator (see figure 1.5 for example)
- Information about the tools used (e.g., versions, links, installation guidelines, good resources)
- The directory organization
- The format of any input files
- The purpose of every file associated with the generator
- The unit test process
- The installer build process
- Information required to explain the special cases covered by the generator
- Behaviors of the generator that would not be apparent at first glance to someone unfamiliar with the generator
- Contact information for the current maintainers
- Known bugs

The second document is aimed at the end user. It describes how the system is to be used and should address the following:

- What the generator does and does not do
- Installing the generator
- Testing the installation
- Running the generator
- Deciding who should run it
- Determining when it should be run

You should also include:

- A warning about altering the output of the generator by hand
- A graphic that shows the input and output files and the flow of the generation process

- An illustration of possible problems (for example, if a runtime error occurs, describe what that error looks like and what it might mean)
- Contact information for the current maintainers

If the end users of the generator are also the maintainers, then you can merge these documents.

Buy it and alter it

If the third-party code is close enough to your ideal but not quite there, you should consider building a code munger to alter the code to your specifications post-generation. Building and maintaining a custom code generator is no small effort, and anything that can spare your company that expense is worth considering.

1.4.4 Deploying it

Deployment is the most critical phase—and often the most overlooked. Does a tree fall in the forest if nobody is there to hear it? The same applies to programs. A program is of no use to anyone if it is never used. You'll want to concentrate on two areas during deployment: creating installation tools and educating your users.

Making the installers

Your generator will go nowhere if it can't be reliably installed or easily used. You should spend the time building and documenting an installation tool for the components required for the generator. In addition, you need a facility for testing the installation without running the generator. This will make it easier for people who are nervous about running the generator to ensure that it is able to run when they want it to.

Educating the users

Software tools, particularly ones built in-house, need evangelism and support. As the builder, buyer, or maintainer of a generator, it falls on you to educate your fellow engineers and managers on the use and the value of the generator.

An easy way to educate a large group of people is to give a seminar. The seminar should be brief, but should at least cover this information:

- Describe what the generator does.
- Describe what the generator does not do.
- Emphasize the value of the target code and its impact on the project. Emphasize that the code that the engineers write is important, and that the generated code is also important. An architectural decision was made to build the code that is output from the generator; you should support this decision.
- Show the architecture. Explain at a high level how it does its job, but leave the gory details to a question-and-answer session.

- Show how the generator is run. Ideally, you should run the generator during the session so that people get a feel for the code generation cycle and how it integrates into their workflow.
- Show the generated code in action.
- Address fears. Fear of the unknown is natural. At this point you will have been working with the generator for a while, so it will be familiar to you. This will not be the case with others. It is important that you address their natural fears about this new tool. It is also important that you do not attempt to teach them every little detail of implementation of the tool. If you drill down to a low level, then your audience won't know if what you are telling them is important information about how to use it, or just self-aggrandizing trivia about how you built the generator.
- Show the future potential without looking like a zealot. This is a tricky balancing act. You want to appear enthusiastic about the potential of the generator without looking like a megalomaniac. The last thing you want is a room full of people who interpret your words about the generator doing this and that in the future as "It will do your job, and your job..."
- Point people to the documentation.

For more information on how to give an excellent lecture, you should read the books of Edward Tufte: *Visual Explanations*, *Envisioning Information* (both from Graphics Press, 1998), and *The Visual Display of Quantitative Information* (also from Graphics Press, 1999).

1.4.5 Maintaining it

Any successful software needs to be maintained. If you are the key implementer, you should maintain the project for a while until a suitable replacement can be found to take over the next maintenance period. In addition, you should strive to ensure that you have access to architectural discussions and decisions that might affect the generator.

1.5 THE BUY/BUILD DECISION

The case study spent no time at all on the buy-or-build decision—and that was intentional. This book is primarily about the design and implementation of code generators, and the introductory chapter should talk to the value of the custom solution. In real life, the buy/build decision is a serious one.

To start, there are more options than just buying or building. Several excellent open source code generators are available on the Internet. For purposes of this section, the decision to use an open source code generator will be coupled with the decision to buy, which brings us back to buy and build. If you like, you can say that the decision is to either develop or to use "off the shelf."

The cost of developing and maintaining software is high, so the decision to develop something internally should not be taken lightly. Often the long-term maintenance costs of software are ignored. This is a mistake; maintenance of an existing successful software tool always outstrips the initial development cost.

You and your company will have to decide for yourselves, but we can offer some pros and cons in both directions.

Advantages of building:

- You own the solution outright.
- You control completely the evolution of the tool.

Disadvantages of building:

- Your company must train engineers to use the tool.
- You must maintain the code base long term.
- You must keep the tool reasonably current with the development tools that were used—tools whose version cycles may conflict with your own release schedule.
- The cost of a developer is very high—much higher than the price of an off-the-shelf package.

Advantages of buying:

- There is no upfront developer time building the foundation elements of the tool.
- You may inherit a user community around the tool.
- You can use the documentation provided with the tool to train new engineers.

Disadvantages of buying:

- The deployment cycle needs to account for the time required to customize the tool to the requirements of the application; this could be significant.
- The tool may not work within your development environment the way you would like.
- The long-term evolution of the tool is out of your hands.

We have made every effort to discover the generators that are available both for purchase and from the open source community. We have put references to these tools in the sections that best relate to their function in all of the code generation solution chapters.

1.6 CODE GENERATION AT ITS BEST

With all of the great things we have said about code generation, why not use it for everything?

- Code generation has a large initial schedule overhead for developing the generator before any useful output is created. Code generation becomes genuinely useful only when it is used to create a reasonably significant volume of work.
- You must consider the stability of the design and the feature set. Code generators are ideal for well-known large-scale problems—for example, database access layers, stored procedures, or RPC layers. When the feature set is not particularly stable, or the design for the implementation is shifting, you should consider some hand-coded functional prototypes before implementing a full solution using generation.
- A single tool is not a panacea. Effective solutions are derived by using a number of heterogeneous tools that are well suited to their specific tasks. Code generation is powerful when used appropriately—and laborious when used in the wrong circumstance.

1.7 TOP TEN CODE-GENERATION RULES

Here is a handy set of rules that you can use when you are designing, developing, deploying, and maintaining your code generator:

1. *Give the proper respect to hand-coding*—You should both respect and loathe hand-written code. You should respect it because there are often special cases integrated into code that are overlooked with a cursory inspection. When replacing code you've written by hand, you need to make sure you have the special cases accounted for. You should loathe hand-code because engineering time is extremely valuable, and to waste it on repetitive tasks is nearly criminal. The goal of your generator should always be to optimize the organization's most valuable assets—the creativity and enthusiasm of the engineering team.
2. *Handwrite the code first*—You must fully understand your framework before generating code. Ideally, you should handwrite a significantly broad spectrum of code within the framework first and then use that code as the basis of the templates for the generator.
3. *Control the source code*—I can't stress enough the importance of having a robust source-code control system. This is critical to a successful code-generation project. If your generator works directly on implementation files that contain some hand-written code, make sure you have a versioning system running that can protect your work.
4. *Make a considered decision about the implementation language*—The tools you use to build the generator do not have to be the same tools you use to write the application. The problem that the generator is trying to solve is completely different

from the problem being solved by the application. For that reason, you should look at the generator as an independent project and pick your tools accordingly.

5 *Integrate the generator into the development process*—The generator is a tool to be used by engineers; thus, it should fit cleanly within their development process. If it is appropriate, it can integrate with the integrated development environment (IDE), or in the build process or check-in process. For examples of how to integrate a generator with an IDE, refer to appendix D.

6 *Include warnings*—Your generator should always place warnings around code that it generates so that people do not hand-tweak the code. If they hand-tweak the code and rerun the generator, they will lose their revisions. In addition, your first response to people ignoring the warnings should be to help them and not to berate them. The fact that they are using your tool is a big step. Learn why they needed to ignore the warnings and improve the generator or the documentation. You are the emissary of your tool.

7 *Make it friendly*—Just because a generator is a tool for programmers doesn't mean it gets to be rude. The generator should tell the engineer what it's doing, and what files it has altered or created, and handle its errors with a reasonable amount of decorum. It may sound silly, but a tool that is difficult to use or that's flaky will be ignored and your efforts will be wasted.

8 *Include documentation*—Good documentation is a selling point for the generator. Your documentation should be thorough but not overwhelming, and should highlight the key points: what the generator does, how it is installed, how it is run, and what files it affects.

9 *Keep in mind that generation is a cultural issue*—Educating your colleagues through documentation, seminars, and one-on-one meetings is critical to successfully deploying the generator. People are skeptical of new things, and a good programmer is twice as skeptical as the average person. You need to break through those concerns and doubts and emphasize that you designed the generator for their benefit.

10 *Maintain the generator*—Unless the generator is just a temporary measure, it will need to be maintained long term. If the generator manages a large portion of code, treat it just as you would an engineer maintaining that same piece of code. Your budget should include dedicated time and money for maintaining and upgrading that resource.

1.8 GENERATORS YOU ARE USING TODAY

Understanding that tools that we use every day are code generators can help ease adoption of code generation techniques. Compilers are the most common form of code generators. Compilers generate assembler or virtual machine operands from a high-level language (e.g., C, C++, Java, or Perl).

The C preprocessor is a code generator that is commonly used. The preprocessor handles the `#include`, `#define`, `#if`, and `#ifdef` precompiler directives. Another common generator is a resource compiler that takes a text definition of application resources and builds binary versions of those resources for inclusion in the application.

1.9 SUMMARY

Code generation is an extremely valuable tool that can have a stunning impact on productivity and quality in software engineering projects. It is my hope that through the rest of this book you'll gain a solid understanding in the principles, design, construction, and maintenance of code generators.

In the next chapter, we introduce the basic forms of code generation.

PROGRAMMING

CODE GENERATION IN ACTION

Jack Herrington

Developers using code generation are producing higher quality code faster than their hand-coding counterparts. And, they enjoy other advantages like maintainability, consistency and abstraction (see inside). Using the new CG methods they can make a change in one place, avoiding multiple synchronized changes you must make by hand.

Code Generation in Action shows you the techniques of building and using programs to write other programs. It shows how to avoid repetition and error to produce consistent, high quality code, and how to maintain it more easily. It demonstrates code generators for user interfaces, database access, remote procedure access, and much more.

Code Generation in Action is an A-to-Z guide covering building, buying, deploying and using code generators. If you are a software engineer—whether beginner or advanced—eager to become the "ideas person," the mover-and-shaker on your development team, you should learn CG techniques. This book will help you master them.

What's Inside

- Code generation basics
- CG techniques and best practices
- Patterns of CG design
- How to deploy generators
- Many example generators

Over his twenty years of development experience, **Jack Herrington** has shipped many software applications helped by code generation techniques. He runs the Code Generation Network (www.codegeneration.net).

"will open your eyes to the potential of code generation ... insightful and practical"

—John Lam
co-author of *Essential XML*

"... an endless stream of ideas on how to automate the repetitive aspects of programming"

—Michael Lenaghan
Frogware Inc.

"Like unit testing, once you try generative programming, you'll never look back. This book will become a classic."

—Richard Rodger, Jostraca

INCLUDES GENERATORS FOR

- Database access
- RPC
- Unit tests
- Documentation
- Business logic
- Data translation

www.manning.com/herrington

AUTHOR ONLINE Author responds to reader questions

Ebook edition available

9 781930 110977 54495
ISBN 1-930110-97-9

MANNING $44.95 US/$67.95 Canada