Applied Software Synthesis

Alexander Sverdlov, Danny Kopec
The Graduate Center, Brooklyn College
The City University of New York

Abstract

Over the years, software development has undergone numerous improvements, but writing software is still considered an expensive, tedious, and error prone process. Compared to manufacturing, software development is still in the pre-industrial era whereby products need to be custom built by hand. Clearly there has to be a way to modernize software development—to build the equivalent of an assembly line for software.

This article presents an approach to building robust, large scale systems with the aid of code generation techniques. It begins with a discussion of 'what can be effectively generated', then proceeds to discuss the benefits and risks involved, along with issues regarding knowledge representation and the design of code generation tools. The article concludes with an overview of real world projects which have benefited from code generation techniques.

1 Introduction

The ultimate silver bullet for computer programmers would be software synthesis from specifications. A user or developer creates a specification (in some unambiguous format), which is then fed into a black-box synthesis machine, which in turn produces a working implementation of exactly what the user needed.

In general, however, this won't ever be the case. There are *essential difficulties* [1] in building software; so while we can simplify or automate certain repetitive tasks, we cannot automate everything. If we could, the system would boil down to writing software via that 'unambiguous' specification, which would essentially be a higher level programming language. Luckily, we don't need much improvement over current methods to potentially realize significant productivity gains. While we may not be able to effectively automate the development of all software, or entire projects, we *may* be able to automate certain parts of certain types of software [2].

2 What is code generation?

In today's world of complex frameworks[1], large portions of many software projects involve repetitive tasks of writing what essentially amounts to boilerplate code. Such as the database code, database access code, networking code, user interface code, etc. Much of the process is non-creative, and lends itself to easy automation [3].

Imagine a hypothetical situation where we are charged with a task of creating an internal information management system for some corporation. After analyzing the requirements, we realize that we'll need upwards of 100 database tables, a lot of business rules, and the system (or some of its parts) will need to be accessible via a native client, via the web, and via web-services; we will also have to worry about a few dozen reports and a very tight schedule. There are several ways of tackling that kind of a project: use 'traditional' development techniques (build everything by hand), or create a code generator that will automate some of the more repetitive tasks.

What can be automated? The database code. Write out the schema in some 'well defined' format (in an XML file). Then write a script to loop through every table definition, and output a 'create table' SQL statement for each one (save it to a `schema.sql` file[2]). But why stop there? We might as well have it output a 'create & update & remove' stored procedures, along with a few default queries (like ones by primary key). Before we know it, from a simple XML file, we obtain a database filled with tables and stored procedures. Using similar reasoning, we may decide to go ahead and generate value objects[3] to store database records. We may then decide to create database access code; for every stored procedure previously created, create an easily callable method that forwards the call to the database stored procedure. Database code is usually wordy, with database connections, type conversions, transactions, and error checking. If we automate that, we'll save ourselves a lot of repetitive typing. Moving on, we may decide to generate network interfaces for every 'object' in our system, maybe even HTML pages and dialog boxes, etc.

In short, code generation is the idea of writing programs that write other programs; sort of meta-programming.

The point of the above hypothetical example was to show that we can apply code generation to a significant portion of an average project, and thus have significant productivity gains, and that it doesn't take much effort to get simple things (like database code) generated.

Notice the apparent lack of concern for any particular programming language, like C++, Java, C#, etc. One of the key benefits of code generation is that it is entirely possible to go from one language to the next without too much hassle (unless we have to worry about non-generated hand-written code).

From a more down-to-earth view, the whole code generation issue doesn't always appear as rosy as described in this section. Building an effective code generator (and the entire infrastructure that goes along with it) takes significant project time and effort. This paper aims to address some of the key principles, and to be motivational enough for everyone to seriously consider use of code generation for their next project.

[1] Java 2 Enterprise Edition, .NET Framework, etc.

[2] The actual file names are irrelevant.

[3] Value Objects usually don't have any code, and act like structures to hold a row of a database.

3 Syntax vs. Semantics

Here, we take a look at various types of software problems. While a lot of the 'average' code is repetitive and non-creative enough to be well suited for automation, there are still areas where automation is unlikely to make much progress. So what exactly can we reasonably hope to automate?

In a quite literal sense, semantics represent the *meaning* of a computer program, while syntax represents the *structure*. From a more abstract point of view, we can perceive these as: *what* the program does (the meaning, the specification, or what business problem it solves), and *how* it does it (the implementation, or programming environment details).

With that in mind, we can view Brooks's *essential difficulties* [1] as software semantics, which we can safely ignore when it comes to code generation. Semantics are hard, and are unlikely to ever get any easier. Examples of semantics would be: problem definition, problem solution, software design, algorithm design, etc.

For code generation, we are left with the syntactic issues; namely the implementation. The reasoning behind this is once we know *what* to do, it is relatively easy to actually do it. This is nothing new [4, 5]. Every programmer learns to design the software before actually writing code. In the words of Larry Wall, the three great virtues of a programmer are laziness, impatience, and hubris [6], referring to the idea that lazy programmers don't jump to code, but take the time to figure out the problem, and then find the easiest and quickest way to implement the solution. Our goal here is similar—to take the time to figure out where we can effectively apply code generation, and then maximize our gain from it.

4 Preliminaries

The absolute first thing to do when embarking on any development project is to determine what technologies or techniques are applicable to the project at hand, and their expected effectiveness. Code generation, like any other technique, *does* have its down sides, and as mentioned above, it is not a true silver bullet. A project must be weighed and all options considered before embarking on a quest of automation. Code generation is not appropriate for all projects, and may actually slow down development. That being said, projects involving networking and databases are *usually* good candidates for massive code generation.

After one decides on the use of code generation, for it to be effective, the software project needs to be managed and properly setup to handle the task. That means a suitable choice of technology and actual implementation of the platform that will host, and provide support for, the generated code. Generated code cannot exist in a vacuum, and it takes a lot of effort to setup the environment where generated code can be utilized.

To start with, a general framework for user logins (authentication), security (authorization), and database access usually[4] needs to be setup. All the required helper and utility classes/components need to be up and running. Once this is in place, work can begin on the code generator that will synthesize code to utilize this foundation.

The market provides many such 'platforms' which are ideal for hosting generated code. A few notable ones are the Java 2 Enterprise Edition, and .NET Framework. The final choice

[4] Assuming a multi-user networked database application.

would obviously depend on the type of application that is being developed. In fact, any type of code can be generated, including HTML/PHP for web-sites, obviously SQL and Stored Procedures, SVG for graphics, and pretty much anything else that appears repetitive.

4.1 Cultural Issues

Before going further, we feel it is important to mention a few cultural issues associated with code generation. Some people working on software development may see code generation as a benefit, while others may see it as a risk. Sometimes their concerns are justified (we can't generate things like algorithms—or code that we can't write by hand[5]), but often they're not ('we can't possibly trust a generator to create our database').

It is *extremely* important that the key developers are comfortable with the idea of code generation. They need to understand how the generator works, its benefits and its weaknesses [3]. People are very good at solving semantic problems (designing algorithms, software, etc.), and we shouldn't de-emphasize that. The last thing we want is for programmers to get the impression that their jobs will be replaced by a code generator.

Often the effectiveness of the generator is uncertain, but there are 'possible large gains' in productivity if it actually works. In such cases, maybe initiating a pilot project involving only a few developers could be a good idea. Many project decisions are a gamble; some work out well, some don't. Code generation is usually a good thing to bet on because of the possible huge payoff.

The previous section describes what may seem like a step-by-step process, but the process usually has all architectural decisions being made simultaneously. Often, the best we can do is to plan on a *possible* code generator (until it can be seen to work, we can't be certain of how well it will perform). Also, not all project decisions favor the code generator. More often than not, it is the code generator that has to adapt to other designs, instead of the other way around.

To sum up the whole issue: software projects are dynamic, usually involving many people with different agendas, and different ideas of how things should be done. On the plus side, writing code generators is usually a lot more challenging (and thus more interesting) than writing something like repetitive database interface code. Most programmers appreciate a good challenge and get bored by repetition. Using code generation to eliminate some of this repetition may lead to a happier and more productive workforce.

5 "Knowledge" Representation

The section title deserves some clarification. We are not referring to 'knowledge' in the general sense. Rather as a way to represent the semantic information of a problem, this may include data, code, templates, etc. Simply saying that it is 'data' is an understatement.

Continuing on with our theme of syntax vs. semantics, we need some way of representing parts of software semantics, to be later used in generating the syntactic portions. The information we care about obviously depends on the type of software we are developing. The actual format of the data is of little importance. The main practical requirement is that

[5]Some non-technical people may see it as a silver bullet that can do anything—even impossible things.

it has to be relatively easy to work with. It may be as simple as lines of text in a file, or data in an XML document, or a whole collection of files involving data and code.

What is important to keep in mind, is that we are not interested in 'all' project knowledge. Only knowledge of a specific part of a particular type of project. A simple example would be to maintain a database schema in an XML file (in a well defined way), and have the generator output SQL code to create database tables[6], and build stored procedures to do common tasks—like insert, retrieve, update, and delete—based on primary keys). While this may be a small part of the whole project, similar techniques applied to many parts of a project can provide significant savings.

5.1 An Example

In order to easily spot the type of 'knowledge' needed, let us go through another hypothetical example; let us imagine we are charged with a task of creating a 'Contacts Book' software. Conventionally, we know that this software will need to maintain names, addresses and phones of people. After careful analysis we decide to create a database schema, database access code, some programming language objects to hold the contact data, and some user interface code. We can now go ahead and actually implement it.

A few weeks after this successful project, we are charged with the task of creating a Patient Info-System for a local clinic. We know that we need to maintain patient names, addresses, phones, and other information. So we decide to create a database schema, database access code, some programming language objects, and some user interface code. We can then worry about actual implementation.

It doesn't require a lot of insight to realize that there are common aspects to both of these projects. Both projects come up with a database schema, then it is mostly a mechanical process to implement the database code, interface code, etc., and user interface code.

So in our quest to generate code, we can start by saving the database schema in an XML file[7], and then writing a small generator to turn the XML file into a database schema definition. After an arduous design process, we may come up with something similar to what is depicted in Figure 1.

Notice that we have created our own names for database types, and that we don't mention anything besides the data we need to store. In a production environment we've noticed that it is beneficial to augment this definition with other things like descriptions (to be used for documentation, as well as tool-tips that pop up under the mouse), field names (the user usually likes to see 'Last Name:' instead of 'Lname:'), etc. Other additions that we've experimented with are more functional, like adding custom querying, and reporting capabilities.

These additions are primarily why this section has 'knowledge' in its title. We are not just interested in semantic information, but also little bits and pieces of things that will help us with future implementation.

Obviously, every generator is different, and what may be useful for one project may be totally useless for another. What generally works well is to consider commonalities between

[6]SQL generation is actually widely used, and is supported by most major CASE/UML tools, like Rational Rose™.

[7]The actual format doesn't matter, but we use XML for simplicity.

```
<meta>
    <object name="Person">
        <attribute name="id">
            <type>int</type>
            <pkey />
            <auto />
        </attribute>
        <attribute name="name">
            <type>string</type>
            <size>32</size>
            <required />
        </attribute>
        <!-- ... -->
    </object>
    <!-- ... -->
</meta>
```

Figure 1: Metadata for Person 'object'.

various projects, and start gathering data that way. Think of what is done in *every* project, then try to automate that process. It also helps to gather ideas from various RAD tools; just considering the information Microsoft Access® or Sybase PowerBuilder® store regarding tables, queries, reports, and forms can provide a lot of ideas about what the code generator should be working with. However, it is important to realize that such tools are designed to be everything for everybody, and are thus usually not focused on any particular problem (which the code generator should be).

5.2 Suggested Format

As can be gathered from Figure 1, whatever data we may be dealing with, XML (eXtensible Markup Language) is generally a flexible and easy format to work with in terms of storing custom data. One of the primary strengths of XML is that we define the format ourselves. If during analysis we realize we need to store some previously unknown information, we just add a new tag to the file. Simple as it is, this format is also very easy to read/write using relatively simple techniques.

5.3 Alternatives

There are of course other ways to approach knowledge representation. In numerous projects we've witnessed the use of UML/CASE tools. These work well for relational database schema definitions (in fact, nearly all these tools allow the user to generate SQL code). Many also

allow the generation of value object[8] code for languages such as C/C++, C#, and Java.

The primary limitation of such tools is that they usually have limited user extensibility. If we'd like to store information that isn't part of the software package, we're out of luck[9]. Another major limitation that's often overlooked is the lack of portability to other environments, or the lack of data export functionality to competitive tools.

Many of these tools allow for custom code generation by letting the user write code in a (usually) proprietary scripting language. From our experience, this approach can work relatively well in 'simple situations', but it can fail miserably when the generation code becomes relatively complex (maintaining a large code base of some proprietary scripting language code can become a nightmare, and makes the project dependent on the few individuals who've mastered the format and APIs).

Overall, these tools try to be everything for everyone. It is certainly better to use one of these for a project than none at all. They are getting better, however, the ultimate flexible solution, that fits the project like a glove, is the one created with that project in mind.

6 Code Generation Tools

As mentioned earlier, we will mostly focus our attention on custom built tools, and not on the tools already available with of-the-shelf UML/CASE software.

So what exactly goes into a code generator? Well, the first thing it needs to be able to do is read the metadata (or 'knowledge' data) file. Store it in memory in some convenient data structure, and then loop through the data to generate, and output source code files of various languages for various parts of the application. That's really all there is to it.

The generation may be template driven, where the user specifies a format for a particular file, which is then generated to that format using the data from the metadata file. On the other hand, a simpler generator may just output code directly (using 'print' statements). In the end, it doesn't really matter which approach is used. The template may be a bit more 'standard' (but may take longer to develop) while a program-driven approach may be easily tweak-able (and takes less time to create).

The choice of programming language is important. It has to be a language the programmer is familiar with. It has to have support for data structures (use of a hash-tables, or associative arrays is quite convenient), have the ability to easily quote code (we don't want to put a backslash every time we want to output a quotation symbol), and be able to do things with relative efficiency. For all these and other reasons, we've mostly used Perl as the language of choice.

It is possible to use other languages. A very clean approach that has been used employed XML to store the semantic information, and the use of XSLT to transform that into SQL and Java code. The entire code generator consisted of XML style-sheet files.

Let us say we've decided to use Perl (keeping in mind that any language can be used), how do we go about generating code from the XML file defined in the previous section? After the XML file is parsed and saved in an appropriate format, the SQL generation may[10]

[8] Structure like objects that store a row of data from the database.

[9] Some packages allow for storing of arbitrary key/value pairs; often this not flexible enough.

[10] Note that this is only an incomplete excerpt.

look something like Figure 2.

```
foreach $object (@{$objects}){
    print $out "CREATE TABLE ".$object->{sqlname}." (\n";
    foreach $attrib (@{$object->{attrib}}){
        print $out "\t".$attrib->{sqlname}." ".$attrib->{sqltype};
        print $out " NOT NULL" if $attrib->{required};
        print $out " IDENTITY(1,1)" if $attrib->{auto};
        print $out ",\n";
    }
    print $out "\tPRIMARY KEY(".$object->{sqlpkey}.")\n);";
}
```

Figure 2: Perl code to generate a SQL 'create table' statement.

The code in Figure 2 doesn't provide the whole picture; just a glimpse of the idea we are trying to convey. There is much more in the background that needs to happen besides this code. For example, nowhere in the XML file (Figure 1) do we have the `sqltype` tag. The idea is that our 'knowledge' file is using language (implementation) independent types (like integers, strings, etc.) which are then converted to database specific types by our generator; this is where `sqlname` and `sqltype` come from. While this code is relatively simple, it is still capable of generating simple SQL statements. Figure 3 presents the output of this code.

```
CREATE TABLE [Person] (
    [id] INT IDENTITY(1,1),
    [name] VARCHAR(32) NOT NULL,
    PRIMARY KEY([id])
);
```

Figure 3: Generated SQL code.

Hopefully it doesn't take much creativity to realize that the code in Figure 2 could just as easily have generated Java or C++ or C# or PHP code; or that it could've generated more complex code (like a stored procedure, or code that calls a stored procedure, etc.)

This is the heart of code generation. It is seemingly simple and very easy to get going, but when we consider the fact that we can multiply the output by 100 times (generate 100 tables, or 100 Java classes, or 100 database interface modules, etc.,) then we really start to realize significant savings, both in terms of morale and typing time[11].

[11] In fact, some [7] have suggested that the initial ratio of 'manufactured artifacts' to specification elements should be 20 to 1, and go up from there.

7 Common Problems

One of the major problems is the realization that code generation is not a silver bullet. That is, it doesn't solve all problems, and that it still involves a significant time investment (in fact, sometimes more than it would take to develop the software without code generation).

It is all about perspective. If one approaches code generation as a solution to all software problems, then he/she will be disappointed. If however, it is perceived as a tool to avoid repetitive typing, it can produce surprising results.

Many software artifacts such as user interfaces, reports, and logic code, cannot be effectively generated. We can generate 'default' bland user interfaces, and reports, but subsequently those almost always need to be tweaked by hand. An effective strategy would allow the user to create 'tweaked' copies of the generated code—copies that are not wiped out on the next run of the code generator. Nothing is more counter-productive than updating generated code over and over again.

Other common problems relate to bad naming conventions. The generated code has to have easily identifiable names. Every generated file needs to have a comment about the fact that it's generated, and from what metafile, and what/where to modify to change the contents of the generated code.

8 Common Benefits

When it comes to code generation, there are far more benefits than problems. The most obvious benefit is time. Once we setup a platform for code generation, adding components is easy. The following highlights some of the other benefits.

Code Extension: From our previous Contacts Book example (Section 5.1), what if we also needed to store a person's birthday? In a conventional programming model, we'd have to modify the code in half a dozen places to get it to work (with the database, networking, GUI, etc.). If code generation is used, then we'd only have to add the birthday attribute to the metafile, and recompile the code[12].

Code Robustness: Code robustness is increased. We write the code once and write it well. Everything that can possibly go wrong is considered and proper error messages are provided. Finally, a duplicate of that code is generated for every object in our system.

Error Repair: Errors in generated code are also easily fixed. Imagine we realize that the database code has a major problem if the user enters out of range dates. Also imagine that we have several dozen dates all over the place. No problem; with code generation, we simply update the code that inserts dates to check for the invalid values. In almost no time, a problem that affects the whole application is fixed in a single place.

[12]We may also need to tweak the user interface, if not using the bland generated one.

Efficient Updates: Similar idea applies to updates. If we want to change the generated code, it is very easy to do so. For example, in a client/server application, we may wish to check for invalid input data on both the client *and* the server. Accomplishing these tasks is much easier if code generation is used.

Project Size: Another major benefit is the fact that considerably less code needs to be managed. Instead of having files upon files of source code, our primary concern is the metadata file that stores the semantic information needed for the code generator. If properly thought out and setup, by adding a few lines to the metadata file, we can generate thousands of lines of code that add new components and functionality to the whole application.

Improved Portability: It is also much easier to port code. To turn our hypothetic Contacts Book example into a web-based application, all we would need to do is rewrite the GUI generator code to output PHP/HTML.

Development Cycle: For the most part, coding boils down to making changes (usually minor) in the metadata file, changes to the code generator, and tweaking the generated code (making GUIs/reports nicer & adding business logic).

9 Economies of Scale

Building an effective framework and code generator requires a lot of time. It is usually overkill for small, one-time projects—it is best applied on large projects or in a situation where many similar projects are being developed.

Manufacturing is a great example. Nobody would build an assembly line to manufacture just one item of a product. Corporations spend the time and money building assembly lines so they can produce many items cheaply. A similar idea applies to code generation. The initial investment may be high—making the technique less than useful for small one-time projects—but it can provide a high pay off for every subsequent project that employs it.

10 Case Studies

Chronologically, the Database Enabled Website (Section 10.2) project was undertaken a few years before the Distributed Application (Section 10.1) project. As can be evident from the breadth of application of code generation, much has been learned in that time span. We are still learning. The 'Pitfalls' sections in these case studies illustrate some of the key code generator related problems encountered during the projects, and some possible hints on how to avoid them.

10.1 Distributed Application

10.1.1 Background

This project involved the re-creation of an information management system for a New York City municipality. The original system was showing its age; its design and technology were relatively obscure by modern standards, and it lacked any sort of documentation, making it next to impossible to extend and modify.

10.1.2 Architecture & Technology

It was decided that this project would be implemented using the Microsoft .NET Framework. The project would be a 3-tier distributed application, with the back-end consisting of a Microsoft SQL Server™, the middle-tier consisting of a .NET Remoting service (we also refer to it as the business-tier), and a front-end (the user client) would be implemented using WinForms client. All the coding was to be done in C#.

The user client would connect to the .NET Remoting service, login, and proceed to call business level methods on distributed objects. Those in turn would use DAOs (Database Access Objects) to call stored procedures on the SQL Server. So all interaction with the middle-tier would be done via .NET Remoting, and all the interaction with the database would be through ADO.NET by making stored procedure calls.

The .NET Remoting calls would accept a serialized value object representing the particular entity we were working with. The DAOs would accept and return those value objects, along with queries, actions[13], and other database related methods[14].

The system also needed to provide custom user authentication and authorization. We decided to recreate the 'standard' user-roles-permissions relationship, where users have various roles, and roles have various permissions (the permissions in our case are the actual method calls to the business-tier).

10.1.3 Implementation

From the very beginning, taking note of the tight schedule and the breadth of this project, we realized that we would need to do things in a more general way. That is, we knew many parts of this project were very heavy in terms of code (the database interactions, the networking, etc.), and that much of it was relatively straight-forward boilerplate code.

Judging from our previous projects, we knew we could generate the database schema, database access code, and .NET Remoting interfaces, and this is where we began with our code generator. After about a week, we were set generating all those artifacts[15].

At this point, we had a metadata file that looks very much like the excerpt shown in Figure 1. It was generating the SQL create table statements, stored procedures to: create, update, delete, select all, select count, the C# value objects for every entity in the system, the DAO objects for every value object (to call every one of the created stored procedures).

[13] An 'action' as we defined it is a query that doesn't return data.

[14] Methods that involve table maintenance, like finding the count of records, etc.

[15] Note the fact that we reused ideas and generator layout from previous projects—otherwise this may have taken a bit longer.

We then moved to the .NET Remoting side of the project, and decided to generate the code for the interfaces, and implementation of networking as well. So for every object in our system, we generated an interface for .NET Remoting that would correspond to our DAO calls (which in turn correspond to our stored procedure calls). At the same time, we implemented security (call-context based global id that is used to manage a user session), which was automatically (thanks to the code generator) incorporated into every distributed call we had in the system. The security would verify that the user is logged in, and that he/she belongs to the correct role that has the correct privileges to call that particular method.

Later, as a two-hour experiment, we implemented bland WinForms GUI interfaces for each object. These would display a list view of objects. When the user clicked on one of these, it would present an update form (which would call the remote update method). There was also a button where the user can create a new record (or delete the existing record). We later made that code (all across the project) nicer by implementing list view sorting, and better error handling on data entry/update forms (if a field is required, in addition to highlighting the field in a different color, the client would display an error message if that field is left blank). Along with generated tool-tips (which came from field descriptions—part of the documentation of metadata, these bland forms turned out to be quite useful, and saved a significant amount of time for 'simple' objects such as the management of States, Countries, and many system specific types.

Later extensions to those forms included drop down lists for relationships (if we're in an address update form, then we'll get a drop down list of states, etc., and if state is required, then the drop down won't have a blank, etc.)

Somewhere around the middle of all that we started customizing queries and reports. We handled that with a customized addition to the metadata file, which included what we're searching for, what parameters the query accepts, what it returns, etc. With about ten lines of code in the metadata file, the code generator has produced:

- A 'query' stored procedure.
- Necessary code in DAO objects to call that procedure.
- The remote call interface and...
- The remote call implementation to perform the query.
- User permission information for that method (so administrator can grant users the right to call that method).

Actions (queries that don't return any data) where handled similarly.

10.1.4 Benefits

The most positive benefit of all this was the huge savings in time. A relatively large project was created on time and on budget. We also got into the 'problem solving' part much more quickly, as opposed to solving the infrastructure problems.

Code quality was very high. Using database code as an example, the code would perform all updates in transactions (nesting them if necessary), would handle null values correctly for every single field, and would generate meaningful error messages (such as "Such and such field is not allowed to be empty.").

A huge benefit was realized when GUI code got involved. Not only could we 'ignore' the simple forms (those that would only need tweaking: resizing, rearranging, etc.), but we also gained a huge amount of useful data in the metadata file that we would normally not think of putting in there. For example, instead of just knowing that we have a field name 'Fname', we also now put in the name to use in the GUI form, which was 'First Name', along with a description (for a tool-tip), which enabled us to have very nice code comments and meaningful error messages for both the business tier and the database.

10.1.5 Pitfalls

There were many pitfalls in the project. The most annoying hindrance (which hopefully we won't repeat) was our decision to use a proprietary reporting tool, which we couldn't generate code for (reports had to be laid out in a graphical tool, and 'compiled' along with the project). To add to the ordeal, since it was pretty easy to update the schema, report queries, etc., we ended up spending a significant amount of time ensuring that the reports worked with the generated code.

10.1.6 Case Summary

Overall, the project was successfully implemented, and is being deployed as this paper is being written. As with any project, there were many issues that needed to be overcome, but without a doubt, this project greatly benefited from code generation.

10.2 Database Enabled Website

10.2.1 Background

This project involved a web-based data management system. It was deployed in an intranet environment, and was primary used by employees to manage nearly all the data (customers, invoices, payments, etc.) the company had. It was a new development effort dedicated specifically to solve data management needs of that corporation.

10.2.2 Architecture & Technology

After doing small proof-of-concept experiments involving EJBs (Enterprise Java Beans), it was decided the J2EE technology was overkill for that project, and a seemingly simpler approach involving Linux, Apache, MySQL, and PHP (LAMP) was decided upon. Everyone would use the system via the web-interface, which would be powered by PHP, which would access MySQL for all the database needs.

The project design called for a rather large number of database tables, and data entry/update forms. Reports[16] would be handled by allowing the user to download plain text

[16]Paper based reports were not the major priority. Electronic reports were the primary objective.

files that list the data.

The web-site would utilize a controller pattern; whereby all requests arrive at a single point (where session, security, and overall page layout is chosen), which then forwards (or includes) the particular code to handle individual user requests. The point of entry for `GET` requests would be `get.php`, and for data submission, or `POST` requests, it would be `action.php`.

Actions (usually data submissions) do not display data. They access the database, usually perform insert or update, and forward the user to a 'get' page where they see the results of their actions.

10.2.3 Implementation

Implementation started without any thought of using code generation. It all proceeded nicely, until the point at which we realized that for every form, table, etc., we were essentially writing the same exact code, except with different names and types. It was at this point that we decided to write a simple code generator to produce the data schema, and database access code.

Moving on from there, we generated bland data entry/update forms and support for simple generated queries. At this point we had many generated PHP files, each doing its own little thing.

We integrated all those forms, etc., by hand to get the final website. It was much easier not having to cut & paste files, but to have most of the file being mostly correct, with the exception of a few minor changes.

10.2.4 Benefits

The primary benefit from code generation on this project was the preservation of morale. Instead of coding a boring form after form, we generated all the repetitive code, and then were able to get on to the more interesting part of building an application using those generated components.

Another, possibly more important, part of the project was that we learned that we can actually generate a decent looking GUI (granted it was just standard HTML, with a hand crafted style-sheets file).

10.2.5 Pitfalls

The major pitfall was the updating of the generated code. In order to integrate the generated forms, we needed to make minor changes to them. If we ever changed the code generator (which was often), we would then regenerate the forms, and be stuck, time after time, making those same changes to the newer generated code.

It wasn't the code generator issue. Had we not used the code generator, then making major site-wide changes would have been next to impossible. But with the flexibility of the code generator comes the hindrance that all non-generated code which relies on the generator needs to be updated as well.

10.2.6 Case Summary

The project was successfully implemented and deployed. The resulting code generator was subsequently updated, modified, improved, rewritten, and reused on many other projects.

11 Conclusion

In today's information age, programmer's time is very expensive. We find that software is ubiquitous, yet the techniques for developing software projects are still relatively mundane in their nature. Any technique which could help make the software development process more efficient could have significant impacts on the industry.

This paper discusses the process of code generation, how it works, some issues involved with its use, and presents real practical applications of it. It was shown that code generation, while having its own problems, is generally a beneficial technique to use during software development.

References

[1] Frederick P. Brooks, Jr., "The Mythical Man-Month", Addison-Wesley, 1995

[2] Krzysztof Czarnecki, Ulrich Eisenecker, "Generative Programming: Methods, Tools, and Applications", Addison-Wesley, 2000

[3] Jack Herrington, "Code Generation In Action", Manning Publications, 2003

[4] Steve McConnell, "Code Complete", Microsoft Press, 1993

[5] Steve McConnell, "Rapid Development", Microsoft Press, 1996

[6] Larry Wall, Tom Christiansen & Jon Orwant, "Programming Perl", O'Reilly, 2000

[7] Fred Wild, *Software Manufacturing*, Dr.Dobb's Journal, #359, April 2004