

Object Model-Driven Code Generation for the Enterprise

William J. Ray

SSC San Diego

Andy Farrar

Science Applications International Corporation (SAIC)

INTRODUCTION

Joint Task Force–Advanced Technical Demonstration (JTF–ATD) was a Defense Advanced Research Projects Agency (DARPA) project in the field of distributed, collaborative computing. In a typical JTF command hierarchy, the critical people, relevant data, and their supporting computers are geographically distributed across a wide-area network. This causes many problems that would not exist if they were all in the same location. The goal of JTF–ATD was to make it easier for people to work together. A system that facilitated the sharing of data and ideas without compromising security, timeliness, flexibility, availability, or other desirable qualities was needed. After experimentation with numerous architectures and implementations, the JTF–ATD concluded that an enterprise solution to data dissemination and access was needed. It also became apparent that the different types of data needed to support JTF missions were as ubiquitous as the missions themselves. Therefore, planning systems would need the ability to associate previously unknown data elements to their plan composition. A distributed, object-oriented design held the most promise to meet these goals.

Unfortunately, building distributed, object-oriented data servers with the complex infrastructure to support enterprise solutions was costly and time consuming. JTF–ATD built the Next Generation Information Infrastructure (NGII) toolkit to address this problem. The NGII toolkit allows developers to code generate object-oriented data servers in days rather than months. The NGII code generator synthesized complex code dealing with concurrency, replication, security, availability, and persistence for each server, thus ensuring that all servers followed the same enterprise rules. The NGII toolkit and its descendant, Quava, are widely used by many projects today to help generate distributed, object-oriented servers with the intelligence to act in concert across the enterprise. Quava is available to the public and can be downloaded at <http://www.saic.com/quava/>.

RELATED WORK

Work related to the topics discussed in this paper includes research in program synthesis, code generation, software prototyping, software reuse, software engineering, and software maintenance.

ABSTRACT

This paper discusses the benefits of using a code generator to synthesize distributed, object-oriented servers for the enterprise from object models. The primary benefit of any code generator is to reduce the amount of repetitive code that must be produced, thus saving time in the development cycle. Another benefit to our approach is the ability to extend the services generated, enabling the code generator to act as a force multiplier for advanced programmers. Having a code generator synthesize complex code dealing with concurrency, replication, security, availability, persistence, and other services for each object server will ensure that all servers follow the same enterprise rules. Also, by using a code generator, developers can experiment more easily with different architectures. One of the final benefits discussed in this paper is that when using a code generator for the data layer of enterprise architecture, changes in software and evolving technology can be handled more readily.

Although much of the research in the fields of program synthesis and code generation deals mainly with optimization, the process of generating code for optimizing digital signal processors (DSPs) or machine language has many similarities to the generation of code for an enterprise data layer. In earlier work, several researchers have generated code from descriptive languages or object models [1, 2, 3, and 4]. Whether the code generated was machine language or code that needed to be compiled is not material to the process of generating the code from a more abstract foundation.

Some researchers even took the generation of code a step further to aid in the creation of control code for multiple processes. In the Computer-Aided Prototyping System (CAPS), code is generated from a more abstract language to simulate a real-time system [5 and 6]. Attie and Emerson synthesized concurrent programs from temporal logic specifications [7].

Software reuse has always fallen short of its lofty goals. The reasons cited for its failure are too numerous to list [8]. Some of the most promising work to help reach the goals of software reuse involves a hybrid approach of program synthesis by making use of reusable code components and code generation [9]. This approach is the one taken by the tools described in this paper.

CODE GENERATOR

Quava provides application developers with an Integrated Generation Environment (IGE) that allows them to convert engineering designs from Computer-Aided Software Engineering (CASE) tools (e.g., Rational Rose, Oracle Designer, etc.) into Unified Modeling Language (UML) encoded design objects. Quava can then generate implementation code that can incorporate Common Object Request Broker Architecture (CORBA), Remote Method Invocation (RMI), Component Object Model (COM), or Java 2 Enterprise Edition (J2EE) services. The developer has complete control over which services, architecture, and language to use for their application.

Design

The Quava system is composed of four basic pieces (Figure 1).

The first piece, the repository adapter, imports data and can communicate with commercial off-the-shelf (COTS) modeling tools, such as Rational Rose or Designer 2000, or read models stored in the Object Management Group's (OMG's) XML Metadata Interchange (XMI) file format. XMI is key to interoperability with other COTS modeling tools. The repository adapter imports a model, which is then instantiated as a UML 1.3 metamodel. Internally, Quava can store its UML

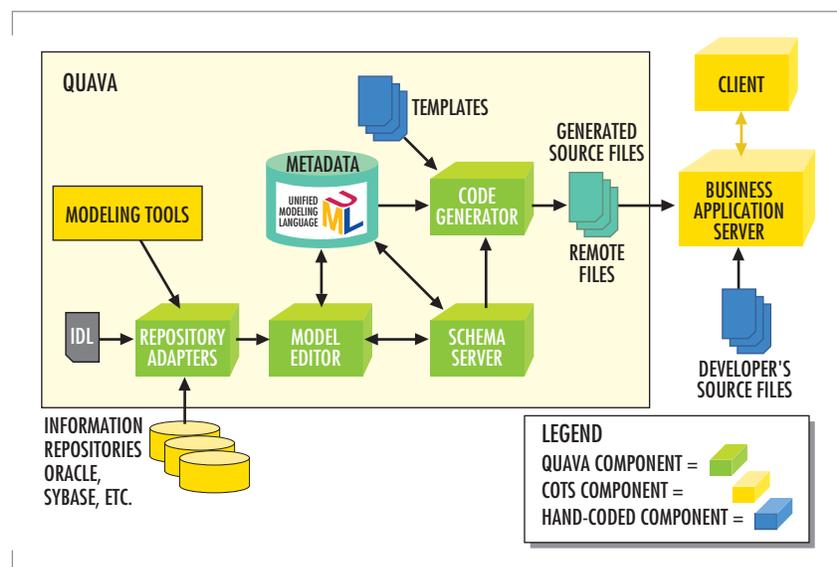


FIGURE 1. Code generation system.

objects either in an XMI flat file or to a UML server, called the Schema Server, for enterprise-wide sharing of models.

The second piece is a tool for altering the UML model. While Quava is not a modeling environment, we did allow for model editing because many COTS tools only support older versions of the UML standard, and many do not support the kinds of additional modeling information designers may want to express. Quava provides the Model Editor, which allows a user to go in and change or add information to the UML model. One example of this is the mapping of one model to another. This is very common when mapping from an application object model to a database model. Model-to-model mapping can also occur between different UML models to automatically generate interface code from a specific model to a shared model. In creating any additional modeling information, Quava still maintains the UML standard by only using UML metamodel objects to represent the additional information. This allows changed models to remain compatible with other COTS modeling tools.

The third piece is a set or sets of templates that guide and direct the generator to precisely what code to produce. Quava differs from many code generators that are used to produce code for a specific COTS tool or environment. Quava users can change or add new templates to allow production of any type of output in any language. The templates are written in either ECMAScript, which is a standardized version of JavaScript, or in Java. The templates allow for maximum flexibility and provide a mechanism for the users to define both the code output and the process flow the generator takes through the model.

The fourth piece is the generation engine. The generation engine pulls in a UML model and then proceeds to apply the selected set of templates against the different elements of the UML model. Processing continues until all selected templates have been processed against the model. Finally, unlike many other tools, the code produced is not tied to Quava in any way and can be imported into whatever development environment the user typically uses.

TEMPLATES

The templates that drive code generation are the key to both the generation's output and the level of control the user has over the generation process. During the course of experimentation with the model-driven code generation approach, we focused on three main issues. The first issue was identifying which types of services lend themselves to model-based code generation. The second issue was how code generation could help with the composition of services in a large-scale architecture, and the third issue was how easily the templates could be extended or new templates added.

Types of Services

To identify which services best lend support to a model-based code generation approach, we focused on where developers spend most of their time. Current software products allow users to generate skeleton code for different architectures, but this code is limited to just a single architecture and does not help with any of the actual logic of the objects. So, where would users get the most "bang for the buck"? Architectural services. Architectural services came to the forefront because they require the

developer to implement additional functionality into each object in the schema in support of the service. For example, an Extensible Markup Language (XML) streaming service may provide a class library for creating the stream and sending and receiving a stream, but the objects within the system will need to implement a method to serialize their attributes to an XML stream. This type of service, where knowledge of the model can reduce the amount of work a developer has to do, is exactly where the code generation process fits in. Below is a very simple ECMAScript template for generating a method to serialize an object to an XML stream.

```

/*****/
// Xml Example/
function writePackage(modelhdl)
{
  var i, interfaceName;
  // Get All the element in this model
  classesList = modelhdl.getOwnedElement();
  // Loop through each element in the model
  for(i = 0; i < classesList.size(); i++) {
    // GLOBAL class object
    xmlClassObj = classesList.elementAt(i);
    // If it's a class then process it otherwise look for nested packages
    if(xmlClassObj.getClass().getName() == "mil.darpa.ngii.uml.umlClass")
      writeClass(xmlClassObj);
    else
      if(xmlClassObj.getClass().getName() ==
"mil.darpa.ngii.uml.Package")
        writePackage(xmlClassObj);
  }
}

/*****/
*****/
/**
 * Write the class structure: header, attributes, and footer.
 */
function writeClass(xmlClassObj)
{
  myXMLFile.writeln("public void writeToXML(StringWriter out)");
  myXMLFile.writeln("{");
  myXMLFile.writeln("  out.write(/"<class>/");
  myXMLFile.writeln("  out.write(/"<classname>"+xmlClassObj.getName()+"</classname>\n/");
  myXMLFile.writeln("  out.write(/"<attributes>\n/");
  writeAttributes();
  myXMLFile.writeln("  out.write(/"</attributes>\n/");
  myXMLFile.writeln("  out.write(/"</class>/");
  myXMLFile.writeln("};");
}

```

```

/*****
*****/
/**
 * Write-out attributes, operations, associations, etc. of a class.
 */
function writeAttributes(xmlClassObj)
{
    featureVector = xmlClassObj.getFeatureList(null);

    for (i=0;i<featureVector.size();i++)
    {
        thisFeature = featureVector.elementAt(i);
        thisFeatureType = new
java.lang.String(thisFeature.getClass().getName());
        if (thisFeatureType.equals("mil.darpa.ngii.uml.Attribute"))
        {
            myXMLFile.writeln("out.write(/"<attribute>\n/");");

myXMLFile.writeln("out.write(/"<name>"+thisFeature.getName()+
</name>\n/");");

myXMLFile.writeln("out.write(/"<type>"+thisFeature.getType().
getName()+ "</type>\n/");");

myXMLFile.writeln("out.write(/"<value>/"+thisFeature.getName()+
+/"</value>\n/");");
            myXMLFile.writeln("out.write(/"</attribute>/");");
        }
    }
}

```

This portion of template code when applied to a simple class:

Class A with attributes:

String name
String address
long age

would produce the following code:

```

public void writeToXML(StringWriter out)
{
    out.write("<class>");
    out.write("<classname>A</classname>");
    out.write("<attributes>");
    out.write("<attribute>");
    out.write("<name>name</name>");
    out.write("<type>String</ type >");
    out.write("<value>"+name+"</ value >");
    out.write("</attribute>");
    out.write("<attribute>");
    out.write("<name>address</name>");
    out.write("<type>String</ type >");
    out.write("<value>"+ address + "</ value >");
    out.write("</attribute>");
    out.write("<attribute>");
    out.write("<name>age</name>");
}

```

```
out.write("<type>long</ type >");
out.write("<value>"+age+"</ value >");
out.write("</attribute>");
  out.write("</attributes >");
out.write("</class>");
};
```

Service Composition

Service composition is the second area we focused on, and it proved to be the most challenging. Composing components within a system is usually a process of plugging in interfaces to well-defined units of functionality, such as Java Beans. Composition of services within an object in a systems schema is much more difficult. We discovered and implemented a number of different ways to compose services without affecting other aspects of the objects although each comes with its own unique issues. The first approach we took was to have the template developer insert calls to outside functions/methods at the correct place in the generation process. This approach, while it did work, did not prove to be very scaleable to a large number of different services because of the knowledge required about each service by the template developer. The second approach was to allow a template developer to implement a set of interfaces, which get calls based on the type of interface or based on template execution. This approach proved to be much more scaleable to a wide number of optional services, but does require the template developer to be much more versed in software development because it currently works only with the Java templates.

Template Modification and Addition

Our third area of focus was the ease of extending and adding new templates. Templates can currently be written in either Java or ECMAScript. Java templates allow for many developers to use the same language that they are using to code their templates. ECMAScript allows developers who have used VBScript or JavaScript to jump in and begin making use of a powerful development tool.

Our conclusion from our work with the code generation template was to concentrate on the Java-based templates. This conclusion was reached based on having the power of a full object-oriented programming language and using the language most developers were familiar with. In addition, because experts in the different areas of software development are usually the people writing templates, they prefer to write in a language that they commonly use.

BENEFITS

Many of the benefits of code generation are obvious, such as the decrease in time to market of new applications and systems, reduction in the amount of new code to be tested, and a reduction in the number of human errors. In this section, we will explore time reduction and some of the other benefits of code generation.

Code generation allows reuse of one of the scarcest resources in most companies: specialized experts. Experts in distributed transactions, security, or concurrence can be used to write specialized templates, thus allowing for corporate capture of that specialized knowledge and providing a force

multiplier to other developers in an organization. Code generation also allows groups to define how they want the code to "look." Styles and enterprise-wide coding standards can be enforced by using templates that follow the standards. Because Quava allows the user to select which sets of templates to apply to their model, developers can experiment with a wide array of architectures and design patterns to see which best fits their specific requirements. Finally, code generation allows developers to be free of their underlying technology. Currently, when a new technology comes out, the developer must go back and re-code an application or system to make use of it. With code generation, new technologies can be merged with current systems, or underlying technologies can be completely replaced by new technology.

Reduction in Development Time

A reduction in development time is the main reason for using code generation techniques. Quava allows the developer to jump straight from the design into the coding phase with very little effort. Normally, the developer is handed a design document and must start from, at best case, generated code skeleton, or at worst case, from scratch. Quava reduces the amount of code a developer must write far more than generators that provide a code skeleton because it is generating object behavior, not just code file structure. Take the example used above for an XML streaming method. This would not be hard to write by hand, but why waste the developer's time doing something that could be generated? A reoccurring benefit of generating methods such as the XML streaming is that any time the model changes, those changes are quickly reflected in the source code. Eliminating human errors that result from typos and simple logic errors also reduces development time. Once a template has been tested, the code that it produces requires far less code testing, allowing the tester to focus more on the business logic of the system.

In our research on code generation, we measured a number of projects with varying object schemas to gather some quantifiable numbers of the kinds of savings code generation could produce. Table 1 shows values captured from some of these projects. The values for lines of code generated have been rounded off to the nearest thousand.

Overall, code generation has been proven to increase the speed at which systems and applications can be implemented, and, with Quava's generation technologies, the reduction is magnified by the experience of the developer.

TABLE 1. Code generation case study.

Case	Number of Classes	Average Attributes per Class	Average Operations per Class	Lines of Code Generated
A	7	30	24	8,000
B	120	22	6	257,000
C	321	12	8	750,000

Force Multiplier

Concurrency issues, complex services issues, and other difficult programming tasks can be encapsulated in templates. By having your best software engineers develop templates, every software engineer that generates an object server with that template may take advantage of their knowledge. In essence, with a software development model where experts create templates and junior programmers develop applications using object

servers generated from such templates, an organization can produce much more high-end software. Of course, the exact value to the organization is only measurable by the number of times a template can be used.

Standardization of Enterprise Rules

By code generating the entire set of object servers with the same templates, a system engineer is guaranteed adherence to these enterprise rules. Different developers can interpret enterprise rules differently. Ambiguities in the software requirements specification can lead to major additional costs later on in the software development process [10].

If developers are allowed to produce object servers with different tools or different templates, it is impossible to guarantee that the system will perform as intended. These differences may even allow for correct execution when the interpretation is constant throughout the enterprise. However, when these different interpretations exist in the same enterprise, errors occur. When the problem domain consists of millions of objects and thousands of object servers, the only feasible solution is to code generate the object servers.

Experimentation

By allowing a system engineer to try different service implementations and middleware without having to encode all of the possible combinations by hand, a system engineer can develop prototypes of multiple test architectures and evaluate their characteristics in realistic deployment environments.

One DARPA project ran into trouble when the deployment environment proved to be less reliable than it was assumed to be. The project used hand-held computers networked with radio waves. When the connections between the hand-held computers proved unreliable, the system performance was severely impacted. Basically, the system would connect to the object servers only to be disconnected by unreliable communications within minutes. The system spent most of its resources establishing and re-establishing connections. The project was able to move from a connection-based architecture using CORBA to a connectionless architecture using HyperText Transfer Protocol (HTTP)/XML by regeneration of the object servers with different templates.

Technology Evolution

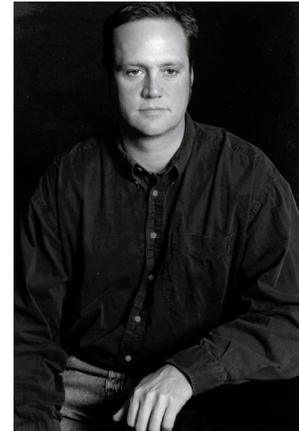
By building your data access and dissemination layer for the enterprise with Quava, your enterprise architecture can handle changes in software technology more readily. When advanced implementations of core services become available, a new template that implements the glue code between the new service implementation and the objects is created, and the object servers are regenerated without having to change any client application software. Also, when new middleware technologies arise, the object servers can be regenerated with additional interfaces so that the object servers can support client applications using the previous interfaces and new client applications using the new interface simultaneously. Older interfaces can be removed when client applications no longer need them by regenerating the object servers without the deprecated interface.

CONCLUSION

In our research, we found that model-driven code generation was a very promising technology with many benefits to the software practitioner. The benefits of using this approach in an enterprise help elevate many of the more substantial problems faced when developing large-scale systems. The openness and flexibility of the Quava implementation gives great support to life-cycle maintenance and software evolution of the system.

REFERENCES

1. Siska, C. 1998. "A Processor Description Language Supporting Retargetable Multi-Pipeline DSP Program Development Tools," *Proceedings on 11th International Symposium on System Synthesis*, 2–4 December, Taiwan, China, pp. 31–36
2. Bringmann, O., W. Rosenstiel, and D. Reichardt. 1998. "Synchronization Detection for Multi-Process Hierarchical Synthesis," *Proceedings on 11th International Symposium on System Synthesis*, 2–4 December, Taiwan, China, pp. 105–110.
3. Leone, M. and P. Lee. 1994. "Lightweight Run-Time Code Generation," *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, June.
4. Engler, D. 1996. "VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System," *Proceedings of the 23rd Annual ACM Conference on Programming Language Design and Implementation*, 21–24 May, Philadelphia, PA, pp. 160–170.
5. Berzins, V., O. Ibrahim, and Luqi. 1997. "A Requirements Evolution Model for Computer-Aided Prototyping," *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, Madrid, Spain, June.
6. Shing, M., V. Berzins, and Luqi. 1996. "Computer-Aided Prototyping System (CAPS)," *Proceedings of the Software Technology Conference*, Salt Lake City, UT, April.
7. Attie, P. and E. Emerson. 1989. "Synthesis of Concurrent Systems with Many Similar Processes," *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, 11–13 January, Austin, TX, pp. 191–201.
8. Lewis, J., S. Henry, D. Kafura, and R. Schulman. 1991. "An Empirical Study of the Object-Oriented Paradigm and Software Reuse," *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, 6–11 October, Phoenix, AZ, pp. 184–196.
9. Bhansali, S. 1995. "A Hybrid Approach to Software Reuse," *Proceedings of the 17th International Conference on Software Engineering Symposium on Software Reusability*, 29–30 April, Seattle, WA, pp. 215–218.
10. Henderson-Sellers, B. and J. Edwards. 1990. "Object-Oriented Systems Life Cycle," *Communications of the ACM*, vol. 33, no. 9, pp. 142–159.



William J. Ray

MS in Software Engineering,
Naval Postgraduate School,
1997

Current Research: Enterprise
architectures; distributed systems;
object-oriented technologies.

Andy Farrar

BS in Computer Science,
San Diego State University,
1992

Current Research: Middleware
technologies; software synthesis;
distributed systems.