A construction process for artifact generators using a CASE Tool

Luiz Paulo Alves Franca
Programare Informática
Av. Americas 500 bl 13 sl 224
22640-085 Rio de Janeiro, RJ, Brazil
Tel: +(55)(21) 2493-5377
luizp@ism.com.br

Arndt von Staa
Departamento de Informática, PUC-Rio
Rua Marquês de São Vicente 225
22453-900 Rio de Janeiro, RJ, Brazil
Tel: +(55)(21) 274449
arndt@inf.puc-rio.br

Background

Since 1992 we are using application and artifact generators. The first application generator was capable to generate fully operational Clipper programs from entity relationship diagrams. Data validation and specific business rules were implemented though escape routines bound to the diagram elements. The generator was build using the meta-CASE Talisman [1]. Diagrams, data dictionary, as well as escape routines were all stored in Talisman's repository. Programs written in Talisman's language generated the application exploring and manipulating its repository. Even though this meta-CASE facilitated the construction of the generator, this approach required highly skilled developers as well as a fair amount of effort in order to code the generating program. In 1996, we used Talisman's facilities to build a generator for Web-based applications that manipulate software metrics databases. During this project, we realized that we could design a generic process to construct artifact generators using meta-CASE Tools. In other occasions we developed tools that generate and maintain C++ and C code fragments and corresponding technical documentation.

Overview of the artifact generator construction process

The generator construction process departs from an example of a simple artifact within the target application's domain. This artifact as well as its specification must be thoroughly verified and validated, since the generator will replicate its structure. The example may be build as a sequence of examples of increasing complexity and scope leading to an incremental generator development and maintenance process.

Given the example, all commonalities (frozen spots) and variabilities (hot spots) are identified [3], as well as the properties that the example's specification must satisfy. The example is then modified in order to contain specific generator tags at all variable points. These tags establish the transformation tags that describe how the specification is accessed and transformed in order to produce the corresponding code within the resulting artifact. The code file enriched with tags is called the *artifact meta-description file*.

A library of transformation rules, written in the meta-CASE manipulation language, must be developed. This library can be built on demand as the conversion from the artifact meta-file to the generator is defined. Typical functions extract data from the meta-CASE repository, adapt them and present them to the generator.

The artifact meta-description file must now be transformed into the artifact descriptor component. This component corresponds to the application domain dependent part of the generator. The independent part is formed by the library of transformation rules and by generation control code. This transformation has been completely automated. A utility program reads the artifact meta-description file and replaces each tag by the corresponding transformation rule. By means of a quite

simple tool the tagged example is transformed into code, which is combined with the transformation library and generation control code, yielding the generator code to be internalized by the meta-CASE tool.

Finally, the meta-CASE must be programmed to adequately edit the specification's representation language. In our examples this is either a modified version of entity relationship diagrams and corresponding data dictionary, or a class structure diagram.

The proposed process has been used to build and upgrade generators that, given a data model, generate a web-based application capable of browsing and updating the corresponding database. The generated applications are able to carry out basic operations such as maintenance of tables extracted from the model, preserving the 1:N and N:N relations defined in the model, including auto-relations, and a simplified form of inheritance. The following generators have been built:

- A three layer client-server architecture: composed of an interface layer (HTML + JavaScript), a business layer (Java servlet), and a database layer (Relational database, JDBC). In our experiments we used both the Talisman meta-CASE and Rational Rose CASE tools as the generator development environment.
- A three layer client-server architecture using Delphi.
- A client-server architecture using VisualBasic.
- A client-server architecture using Centura/SQLWindows.
- A html generator of the specification documents.

Benefits of this approach

- Due to the use of a meta-CASE tool, most of the development effort usually needed to build the input components of the generator (e.g. the application specification editor) became almost negligible.
- Since the construction of the artifact generator is based on a simple and correct example, it has been quite easy to identify the set of transformation points and the nature of these transformations assuring the exact reproduction of the example artifact from its specification. It also became easy to determine the exact composition of the specification. Thus, the specification will not contain fields that are not necessary, reducing the amount of work when specifying an artifact. By means of a reasonably simple set of tests it has been possible to verify whether the generator generates correct code for all cases within the established application domain.
- The artifact meta-description file reduces the impedance-matching problem between the specification and implementation domains. In this file, transformation tags mark specification dependencies. The remainder of the file corresponds to frozen spots, which should be simply replicated in the target artifact. The artifact meta-description file is also a good means to verify by inspection the correctness and completeness of the derived generator.
- The generator replicates the artifact meta-description file rather than the pure code file. Using a simple tool the tags can be removed yielding correctly compilable code. Hence to evolve the generator it is necessary only to add the expanded example's code and adequately tag it. By means of the generator constructor tool a new version of the generator can be constructed. This same process can be used to correct the generator in case the example or the original tagging are found to be incorrect.

- The first experiment led to the creation of a library of transformation rules. Due to the possibility to reuse this library, in subsequent experiments the need to create new rules eventually ebbed out.

Problems found with this approach

- Transformation rules depend on the CASE Tool scripting language. In our experiments, when adapting the generator to use Rational Rose, we had to convert the whole library of transformation rules originally written in Talisman's language, to a new library written in Visual Basic. We also had to modify the utility program that converts the artifact meta-description file into the generator component.
- The syntax of the transformation tags was designed to facilitate the transformer utility, reducing the readability and shareability of the artifact meta-description file.
- In order to increase maintainability of the generated artifacts, version information should be part of the repository. This would allow, for example, to compute the difference between data schemas, allowing the correct and complete generation of database structure updating programs.

XML solution

Currently we are examining the adoption of XML/XMI [4] and Java to solve the problems found in our approach. Since most of the modern CASE tools are capable to generate XML/XMI files from their repository we believe that we can achieve partial tool independence. Total independence may not be possible since some tools may prevent the use of escape code. In any case we expect that the library of transformations rules will require very little changes when adapting the generator to a new CASE tool.

Furthermore the transformations tags should be defined using XML syntax, hence adding rules to the DTD. This approach will permit an artifact meta-description file to be edited through any XML editor. It also assures that generator code can be build for different programming or scripting languages using a same XML DTD.

References

[1] Staa, A.; *Manual de Referência: Talisman: Ambiente de Engenharia de Software Assistido por Computador*; Rio de Janeiro, Brazil; 1993; (in Portuguese).

[2] Hohenstein, U.; "An Approach for Generating Object-Oriented Interfaces for Relational Databases"; In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools*; 2000; pp. 101-111.

[3] Cleaveland, C.; "Building Application Generators"; *IEEE Software* 5(4); 1988; pp. 25-33.

[4] Cleaveland, C.; *Program Generators with XML and JAVA*; Prentice-Hall, 2001.