Code Generation Network Interview with Fabrizio Pugnetti

CGN: Fabrizio, thanks for agreeing to this interview. I wondered if we could begin by asking about your background, how long have you been involved in software development? What types of project have you worked on?

Fabrizio: I've been working in software development since 1982. I started with hardware and software together, actually… I was nineteen and very interested in the microprocessor area, so that I decided to build a microcomputer entirely on my own, without purchasing any "kit".

After that I started working in industrial automation systems for a company based in Milan, then I've been a software consultant for some years. As a consultant, I've learned about modelling languages. Rumbaugh OMT, Shlaer-Mellor and Booch were in vogue at that time. I was very interested in that area, modelling complex applications with those languages was much better than trying to manage them merely at the code level. Since modelling with pencil and paper wasn't the best, I started using modelling tools. Then I've been working for some tool vendors, both in the development and application areas. Finally, the ARTiSAN expansion in South Europe gave me the chance to start working for the Italian team.

CGN: Can you tell us a little about the product(s) and projects you're working on with ARTiSAN?

Fabrizio: We work in several areas around model automation and code generation. This includes both product development, from requirements to test, and research activities.

Research is focused on defining and synthesizing new formalisms and methodologies for generating code from models. Those formalisms have to be easy and pragmatic from the user point of view, but they still need to be based on a solid theoretical model. That's why a good part of our job consists of creating several prototypes and simulating the user working with them. Other research areas are related to synchronization of model objects with non-model objects other than code, e.g. SysML objects and user's documents.

Products I work on are mainly Automatic Code Synchronizer (ACS) and Template Development Kit (TDK), the basis of the ARTiSAN strategy for code generation. On the top of these, we deploy Code Generation, Model Driven Architecture and Design Pattern implementation. Our two scheduled sessions at Code Generation 2007 are intended to give a live demonstration of those capabilities.

CGN: What's the history behind the product? How and when did development start?

Fabrizio: The Shadow technology idea (whose result is what we call today ACS and TDK) was born in the summer of 2003 at a restaurant in Milan. Some people from ARTiSAN UK were in Italy and during dinner we were discussing the best strategies

for code generation. At a certain point, we had an idea; it seemed feasible and we concluded we had to investigate a bit more. The idea consisted of two parts. The first was to offer the code generator to the market *packaged as a UML model*, in such a way end users could encode their own design standards for imposing the rules of their software development process. The second part was to make all that available in such a form that users could click on the tool and get immediate feedback on what is being generated in the code. That's the reason why we adopted Shadow as the technology name, something that is able to follow closely the user but still in a transparent way.

Seven months later, the first version of the product was being presented internally.

CGN: How and when did you get involved? What's your specific responsibility within the team?

Fabrizio: I've been involved since that night, so – definitely – from the beginning of the story. The Italian development team is responsible for developing the ACS and TDK products, related marketing material, research of new code generation technologies and prototyping new model driven solutions, i.e. the code generation products ARTiSAN will offer in the coming years.

CGN: What types of organisation typically use the tools and what sort of problems are they tackling?

Fabrizio: Most of our customers are in the military, aerospace, telecom, automotive and industrial automation segments. These market segments have different constraints, determined by several factors specific to their problem domains.

In military and aerospace the dominant factor is reliability. Thanks to ACS and TDK users in these sectors can automatically re-apply consolidated solutions and reach the end of their projects faster. The automated software factory provided by ARTiSAN also increases engineers' degree of confidence, so simplifies the procedures for the certification processes those companies always have to comply with.

In automotive and telecomms companies time-to-market constraints dominate. With ACS and TDK they can avoid redesigning similar solutions each time; ACS/TDK allow a full reuse of corporate *know-how*, so that engineers can concentrate on *evolving* the technology instead of just *porting* it from the previous project, which is always a very frustrating, time-wasting and error-prone task.

Industrial automation companies have several families of products. Products belonging to a family are very similar – but not equal - to others. Their software shares a common layer plus a delta specific for each product. MDA – implemented by means of ACS and TDK – is the ideal solution for coping with this complexity connected with managing common and specific layers, organized into a set of different platforms.

CGN: What are the immediate and longer-term plans for the products?

Fabrizio: With the next 6.2 release of ARTiSAN Studio (May 2007), we will provide a mechanism for simplifying design and deployment of multi-language, distributed

applications. The MDA paradigm will be used for enabling an application to be arbitrarily distributed onto different hardware nodes connected in a network. Each node can run a piece of software encoded with a given programming language integrated on top of a specific Operating System. Languages and OSs can be different across nodes, even within the same application model.

Studio 6.2 will also enable software engineers to work in parallel but in isolation. In a way similar to how Configuration Management tools work for code files, Studio will allow user to define a *private sandbox* at the model level. That is, operations such as branch, merge, reconcile and rebase – currently only available at the code level – will be offered directly at the *model level*, dramatically reducing the overhead designers have in managing the code base.

The major factor distinguishing ARTiSAN from most competitors is ARTiSAN have always played a role of a *technology innovator*, not a *follower*. Disclosing medium and long-term plans is a very high-risk practice in our market, so I can't say much more. But I can summarize by saying that on one hand we expect new formalisms for addressing the code generation problem; on the other hand, we expect the model-based approach to expand its power into new areas such as model-driven testing; model-driven document generation, and so on.

CGN: Is there anything unique or controversial about the products' approach?

Fabrizio: Yes, many aspects are absolutely unique.

First of all the ARTiSAN code generator is a piece of software that users can manipulate themselves (they *can*: they *don't have to*). Predefined generators are supplied for many implementation languages, but the power of the methodology resides in enabling customers to *encode* their *software development process* directly *into the ACS code generator*. This is absolutely unique.

The ARTiSAN code generator looks like a UML model. No surprises here: a code generator is a piece of software, and modelling software with UML has many benefits. Since ARTiSAN is involved both in UML modelling and code generation techniques, deploying the code generator as a UML model is really a natural choice. Nevertheless, this aspect is still unique: it's like you are in a park full of people, and you realize a £100 note is close to your feet. The first thing you think about is why no one else saw it before. Well, the fact is other tool vendors don't offer code generators as UML models, as *we do*. Specific modelling constructs and an on-purpose action language allow an easy, component-based definition of the generator features. Changing the generator model immediately makes ACS load and use the new DLL without user intervention. In a few seconds users can check how their code generator modification is reflected in the generated code.

The short delay between modification and outcome results in a short learning curve and enables a big productivity increase with this new methodology. This close-loop feedback is unique to the way TDK and ACS have been designed.

Finally, ARTiSAN Studio is able to trace the most abstract SysML requirement forward to the specific line(s) of code that implement it in the software. The *whole*

system and software development process fits within *one single tool* with ARTiSAN Studio.

CGN: Your products are capable of '*building and reusing code generation patterns'* can you say a bit more about what this means, how it works and how it benefits end-users?

Fabrizio: First of all, we have to distinguish between *code generation patterns* and *design patterns.* They are related entities, but strongly different things.

A *design pattern* is a model fragment solving a general problem that can be encountered in several different projects. Better definitions exist but this describes the essence of the idea.

A *code generation pattern* is a component of the code generator, dedicated to implement a given feature. For instance, the *requirement tracing generator pattern* implements a feature that is able to trace high-level requirements to the associated code lines. Users associate requirements to code lines in Studio and the result is that generated code has comments containing the requirement IDs on code lines associated with those requirements. Conversely, if users add a similar comment mentioning a requirement in the code file, a link is automatically created in Studio between that code line and the requirement.

Features implemented by code generator components (or, in other words, *code generator patterns*) belong to a number of possible areas: code instrumentation for debugging and testing, tracing, code formatting *and design pattern implementation*. A code generator feature *can* implement a design pattern, simply by dragging stereotypes to the relevant objects in the model editor. This is the connection between the *design pattern* and the *code generation pattern*: a code generation pattern can be responsible for implementing a design pattern (but the opposite is not true).

The benefits of this approach is clearly the possibility of reuse. The dream of all software engineers is to be able to *reuse their ideas*. Due to some historical and common practices, it tends to be hard to introduce further requirements into existing software. Unlike other fields of engineering, if we have a software application "A" able to do "A" and another software application "B" able to do "B", we *still don't have* a software application "A+B" able to do "A+B". In most situations, the fact that we have separated applications "A" and "B" is irrelevant for solving the "A+B" problem. Maybe the "A+B" application would contain some code lines from "A" and "B" together, but in the general situation there is *not* any practical way to *automate that sum*, and no way of saying in advance how much the cost to develop the sum is.

That's the reason why the software community has developed a number of ways of solving exactly that problem. Object orientation, generalization, templates, patterns and UML modelling are all antidotes to the difficulty of modifying software to handle new requirements.

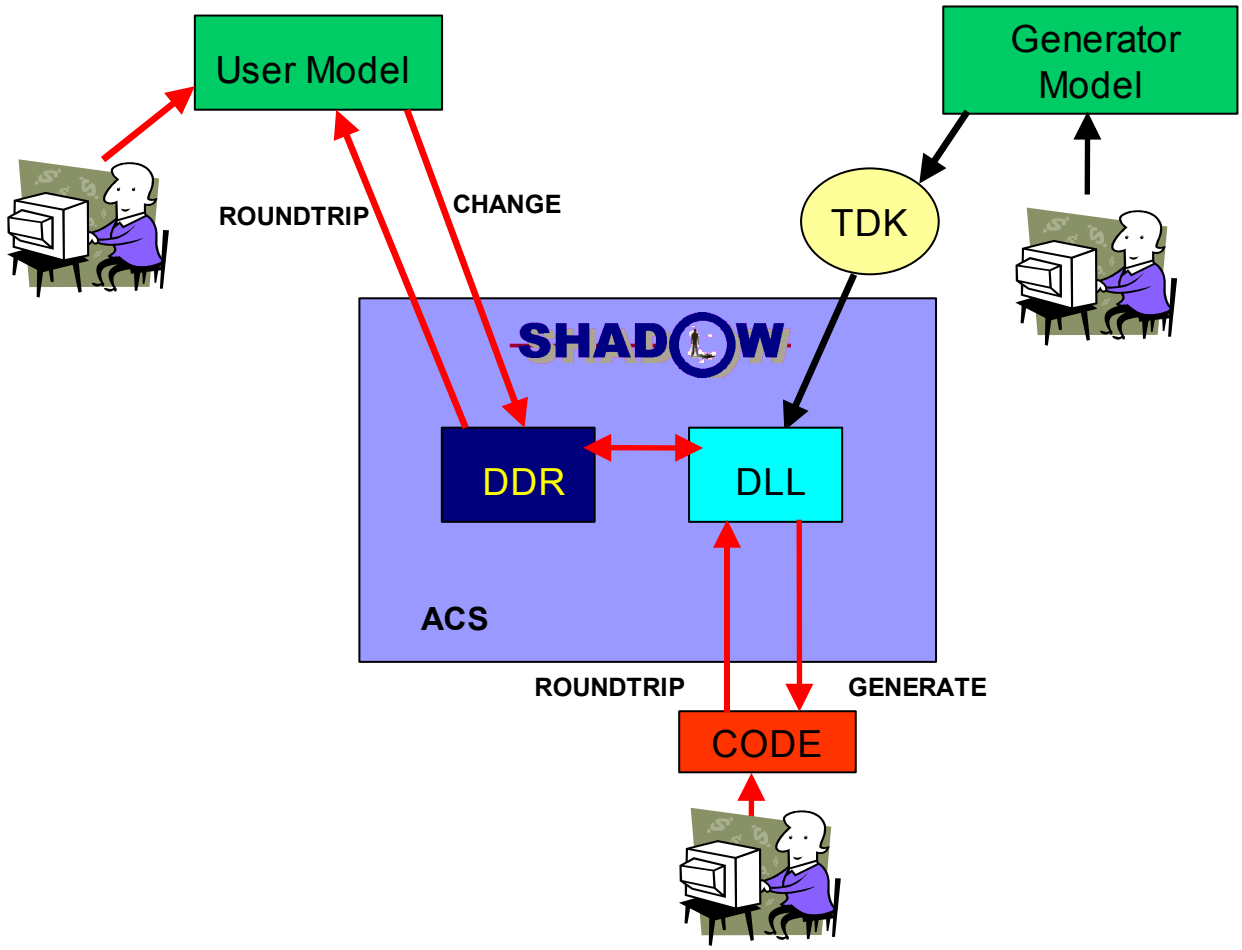
Code generation patterns are an effective way for reusing ideas as well as already experienced solutions within the same project or across different projects. Each reused idea or solution is encoded as a *code generation pattern* into the code generator itself.

Multiple patterns can coexist at the same time within the same generator, as well as being shared by multiple generators dedicated to different projects.

Due to the way they are defined and implemented, code generation patterns are intrinsically "loosely coupled entities"; this means they don't interfere with each other. So, given pattern "A" implements feature "A" in generated code, and pattern "B" implements feature "B", what designers can obtain by enabling "A" and "B" patterns at the same time is effectively generating features "A+B" together.

CGN: Can you give our readers an overview of how the tools are put together?

Fabrizio: The figure below summarizes the way ACS and TDK work



There are two UML models: the *User Model* and the *Generator Model*.

The *User Model* is the Studio application model representing the project we want to develop.

The *Generator Model* is a new concept that enables us to change the way the *User Model* is transformed into code by the code generation process.

So changing the User Model changes the generated code, while changing the Generator Model changes *the way (the algorithm through which) code is generated*.

In the preceding figure the Shadow rectangle contains two main components:

- DDR (Dynamic Data Repository), and
- DLL (the Code Generator DLL).

Shadow ACS reads the User Model into the internal DDR - that is, a fast, memory-based repository suitable for use in the code generation phase.

Each time a difference is discovered between the User Model and the DDR, that difference is first loaded into the DDR, then a generation action is performed.

"Generation" is composed of two distinct steps: first a *model-to-model* transformation is run; in MDA terms, this is the step responsible for transforming the Platform Independent Model into the Platform Specific Model. The second step is a *model-to-text* transformation, i.e. A transformation of PSM into code. Both steps are achieved by inspecting the model contents programmatically using algorithms residing in the DLL. These same algorithms are held in source form in the Generator Model, and are translated to fast executable code (the DLL) by means of a special compiler contained in the TDK block.

When the User Model is modified (e.g. a new item is added by the Studio model editor), Shadow ACS detects the change and generates the code.

In addition, when the Generator Model is modified, the following occurs:

1. TDK detects the change and generates corresponding generator code expressed in the object oriented template language (.sdl)

2. TDK transforms the sdl code into an intermediate language that can be compiled (C++)

3. TDK compiles and links the intermediate language files and builds a new version of the Code Generator DLL

4. Shadow ACS detects that a newer DLL is available than the one currently loaded; so it replaces the loaded DLL with the new one, and regenerates the code

The process above takes a few seconds in order to complete. The effect is that the user quickly sees the results of the changes in both the User and Generator models in terms of the generated source code. This quick feedback provides a very effective 'rapid prototyping' environment for the development of new code generation features.

CGN So how do customers get started with the toolset?

Fabrizio: The ways customers can start using the product varies a lot from one organization to another. But once customers start using the products, they usually quickly discover new application areas. So, after a basic training course about the generalities of the tools, we often get requests for specific kinds of training, related to tool use within a particular environment.

Legacy code bases can be imported into Studio with a set of *code to model reversers*. Each supported language has its own flexible reverse utility able to generate a model from legacy code.

CGN: How long does it take to master the product? Does it involve a radical change of mindset on behalf of the developers?

Fabrizio: Our customer's experiences show very quick adoption. This is the expected result, indeed: every detail of the product has been designed on purpose for shortening the learning curve. The fact that users can quickly see the results of their model-level changes at the code-level results in very quick feedback on their learning.

I must say that in 90% of cases, ACS and TDK have been sold after just one practical demo of these products to technical people. This is proof that *people were just waiting for this*. That's to say that the technology doesn't require a radical change of mindset from developers.

CGN: Do you know of other organisations that produce similar products?

Fabrizio: OMG prescribes very precise rules about UML constructs but doesn't say anything about the way UML should be mapped to code. UML tools are differentiated in many ways: each vendor offers a distinct set of features, regarding for example ergonomics, user interface, toolbars and add-ins such as document generation, integrations with other tools, and so on. However, in the end, UML modelling of use cases, state machines, classes, etc. always have the same *unified* meaning inside every environment. This is due to the existence of the OMG standard that carefully specifies the meaning of each entity.

Since there is no standard defining the mapping between model and code, we can expect that functionalities offered in the code generation level are dramatically different between different tools, And, as far as I know, we are the only ones offering a code generator environment having the above set of features, allowing engineers to encode their software development process directly into it.

CGN: Are you working on any other projects that may be of interest to our readers?

Fabrizio: There are many projects under way, all related to MDA. One of them that is very promising, is the *Template Wizard*, an ACS/TDK add-on enabling the implementation of *design patterns*.

This tool allows the whole design pattern life cycle to be handled. It is possible to *prototype* a new pattern, just by entering the pattern's UML objects into the model editor directly. After that user can *develop* the pattern, adding any required special

case handling and other customizations. Finally, the pattern can be *debugged and tested* in order to finalize its functionality. Modelled patterns can then be *applied* to real objects with a simple formalism with the resulting production code generated automatically. The whole process is interactive as usual, for each “click” users can constantly verify the result of how their choices are reflected at code level.

CGN: Thank you for your time Fabrizio. I am sure the interview will be of interest to our readers.

Fabrizio Pugnetti is a Senior Consultant at ARTiSAN Software Tools. With over 20 year’s industry experience, Fabrizio has worked in systems and software development in fields such as telecommunications, automotive, medical equipment, industrial automation, etc.

For more information on ARTiSAN's tools follow the links below:

ACS
TDK

© Code Generation Network, Fabrizio Pugnetti 2007