Design Pattern Application: Pure-Generative Approach vs. Conservative-Generative Approach

Hervé Albin-Amiot [1], Yann-Gaël Guéhéneuc [2]

École des Mines de Nantes
4 rue Alfred Kastler
BP 20 722
44 307 Nantes cedex 3
France

[1] Soft-Maint
4, rue du Château de l'Éraudière
BP 72 438
44 324 Nantes cedex 3
France

[2] Object Technology International Inc.
2670 Queensview Drive
Ottawa, Ontario, K2B 8K1
Canada

{albin, guehene}@emn.fr

1. Context and Problems

Since their emergence, design patterns have been widely accepted by software practitioners. Their contribution covers the definition, the design, and the documentation of software. Design patterns describe solutions to recurrent architectural problems. Design patterns are meant to improve certain software quality characteristics. For example, the design patterns proposed in [Gamma95] are defined to improve the flexibility and the understandability characteristics of software quality.

For our discussion we summarize software developments in two kinds: Development of large applications (such as accounting application or billing systems); and, development of large framework (such as window system [SunAWT] or networking system [Zweig90]). In one hand, when developing large applications, the developers need to abstract the functional and non-functional requirements of the application. They must be particularly careful with the specifications of the business rules and policies of the application, while the language and the code implementing the application are not really important. On the other hand, when developing large frameworks, the developers need to abstract the potential extension and implementations of the framework. They must be particularly careful with the architecture, the design, and the implementation of the framework, which is, at least, as important as the functionalities provided.

In this position paper, we present two tools that help the developers in implementing large applications and large frameworks, using design patterns. Scriptor [Scriptor] (Section 2) is an industrial-strength application generator. Developers use it to generate large applications from scripts stating their functional and non-functional requirements. PatternsBox [PatternsBox] (Section 3) is an academic conservative application generator. Developers use it to implement design patterns in existing applications. PatternsBox modifies or creates only the required code artifacts (class, interface, fields, methods) to implement the design patterns, leaving the rest of the code untouched.

In Section 4, we present the advantages and the limitations of the two approaches, and a comparison of these with respect to the use of design patterns.

Finally, in Section 5, we conclude and present some future directions, which consist in combining the pure-generative and the conservative-generative approaches.

2. Pure Generation: Scriptor

Scriptor is an industrial tool developed by SOFT-MAINT (SODIFRANCE group, Nantes, France).

Scriptor is a tool to define and apply generative scripts on a model. Scriptor is based on an interchangeable meta-model, for instance, the UML meta-model. The entities and relationships defined in the meta-model are reified as Java classes. Textual scripts and Java actions can be defined and bound with each of these classes. After the tool has loaded a model, it can instantiate the previously defined classes and apply the scripts, for example, to generate code. The tools defines:

- An interchangeable meta-model, the best known being the UML meta-model.

- A WYSIWYG interface to define scripts, which can execute Java-defined actions. The scripts and Java actions are associated with the entities described in the meta-model.
- A module to load models from an Integrated Software Engineering Environment (ISEE), like Rational Rose, Paradigm+, or any other ISEE supporting XMI.
- A generation module to apply scripts on a model. The result may be the code or the documentation corresponding to the model. This module uses a pure-generative approach with a user-defined tag mechanism: The developers must carry out each modification in the ISEE and re-import the model in Scriptor before performing a new generation. If the developers modify the results of the generation by hand, outside of the tags, and perform a new generation, their modifications are lost.

There is no specific support for design pattern application in Scriptor (patterns are described in the scripts), however design patterns can be applied in a systematic way or in selected cases using tags in the source model.

In Scriptor, the reference is the model.

3. Conservative Generation: PatternsBox

PatternsBox is an academic tool being developed at the Computer Science Department of the ÉCOLE DES MINES DE NANTES, France.

PatternsBox is a tool to instantiate design patterns. The term instantiation is commonly used to identify the task of adaptation and implementation of a design pattern solution in a particular context. The tools defines:
- A meta-model, which is tailored for the definition of design patterns, with respect to the instantiation and the detection aspects.
- A repository of design patterns. The design patterns are first-class entities, described in terms of the meta-model.
- A user interface to select a particular design pattern, adapt it to a specific context (number of actors, names of the actors, relations, cardinality).
- A source-to-source transformation engine, (JavaXL). A design pattern knows how to instantiate itself. Using this knowledge, the mechanisms associated with the meta-model and JavaXL, a design pattern is able to instantiate itself in a given source code. The main interest of using a source-to-source transformation engine is the conservation of the code previously written by the developers (comments, layout, structure, idioms,…). The

instantiation mechanism is a conservative-generative solution to the instantiation of design patterns.

In PatternsBox, the reference is the user source code.

4. Comparison of the Two Approaches

With the respect of our first preoccupation, which is the application of design patterns, this section is an attempt to summarize the advantages and the drawbacks of the two aforementioned generative techniques.

The main interest of a pure-generative approach, such as proposed by Scriptor, lays in the hiding of the generated code. Developers never need to look at the instantiated code. They define the functional and non-functional requirements of the application using scripts and UML diagrams, and they generate the source code from these scripts and diagrams. Developers implement design patterns directly in the UML diagrams, for instance, by explicitly stating that a class must follow the Singleton pattern. They are not interested in the code associated with the design pattern. Thus, they rely completely on the tool to generate the equivalent code and there are no sophisticated parameterization capabilities at the scripts and diagrams level.

In a pure-generative approach, the developers must rely completely on the tool to generate the correct and most efficient implementation of a design pattern. Once the code is generated and released to the customers, there is no way to track down what design patterns have been applied and where they have been applied.

This approach is particularly efficient to provide software at low cost by helping developers to focalize only on the business logic of the application. However, if code artifacts or other implementation aspects have a real importance, like during the development of a framework, it seems that this approach is inadequate.

The main interest of conservative-generative approach, such as proposed by PatternsBox, lays in the conservation of all the attributes of the source code. Developers explicitly choose the artifacts in their source code that play a role in a design pattern, adapt the design pattern to these artifacts by parameterization, and then instantiate the design pattern in their source code. No other change is performed on the source code. However the developers must identify the artifacts on which to apply a design pattern. They must take care of the conflicts that arise when applying a design pattern

on an artifact that cannot comply with the desired design pattern. This process is less robust and less productive but more efficient than the one using a pure-generative technique.

It seems that for developments where implementation techniques have a great importance, like for the definition of class libraries and frameworks, conservative-generative approach is really interesting because it provides a good granularity in the interaction with the developer during development stage. In other cases where the details of the implementation is less important than the functionalities provided, this approach is unnecessary costly and even dangerous, if the model of the application must be the only reference.

5. Conclusion and future

Code generation is a very promising technology for the development of quality code. Code generation eases the whole development life cycle by transferring the effort of writing quality code to the effort of defining high-level abstractions, such as design patterns, UML diagrams, and generation scripts.

However, pure-generative approaches and conservative-generative approaches have drawbacks: In a pure-generative approach, the developers have no or little control over the code generated. The code generated may be of good quality with respect to certain quality characteristics, it is not generated to be read by humans, and it does not provide cognitive to help the developers understand the code without the associated scripts and UML diagrams. In a conservative-generative approach, the developers need to write most or a great part of the code by hand, while meticulously defining, implementing and documenting the design decisions mangled with the source code. Then, the developers can improve certain quality characteristics according to their knowledge and understanding of the code.

A solution would be to combine the two approaches, by improving the cognitive and the documentation aspects of the generated code and by helping to refine the source code gradually (without loosing the modifications made by the developers).

6. Bibliography

[Gamma95]   E. Gamma, R. Helm, R. Johnson, J. Vlissides ; Design Patterns : Elements of Reusable Object-Oriented Software ; Addison-Wesley Professional Computing Series, 1995.

[PatternsBox]   Information available at: http://www.emn.fr/albin/

[Scriptor]   Information available at: http://www.qualitec.fr/scriptorUS.htm

[SunAWT]   Information available at: http://java.sun.com/j2se/1.3/docs/guide/awt/

[Zweig90]   J. Zweig, R. Johnson ; *The Conduit: A Communication Abstraction in C++* ; Proceedings of the C++ Conference, pp. 191–204, 1990.