Polymorphic Templates

A design pattern for implementing agile model-to-text transformations

Gábor Kövesdán, Márk Asztalos and László Lengyel

Budapest University of Technology and Economics, Budapest, Hungary
{gabor.kovesdan, asztalos, lengyel}@aut.bme.hu

Abstract. Model-to-text transformations are often used to produce source code, documentation or other textual artefacts from models. A common way of implementing them is using template languages. Templates are easy to read and write, however, they tend to become long and complex as the complexity of the metamodel grows. This paper proposes a design pattern that allows for the decomposition of complex templates with branching and conditions inside into simpler ones. Its main idea is that the code generator does not know about the concrete templates that are called: they are determined by the objects of the model being traversed. The concrete template is selected through object-oriented polymorphism. The pattern results in a flexible code generator with simple templates, good extensibility and separation of concerns. This agility facilitates the design for extension and changes, which is paramount nowadays.

Keywords: Modeling ▪ Domain-Specific Modeling ▪ Model Transformation ▪ Code Generation ▪ Design Pattern

1 Introduction

This paper presents a design pattern that can be used to create flexible model-to-text (M2T) transformation. The pattern is applicable on complex M2T templates and proposes an object-oriented decomposition for the generator, the model objects and the templates. This decomposition achieves reduced complexity, separation of concerns, improved readability and most importantly improved maintainability and flexibility. We believe that the *Polymorphic Templates* pattern will greatly help developers of all kinds of M2T transformations in designing robust code generators that are easy to maintain and extend.

The rest of this paper is organized as follows. Section 2 briefly explains the foundations of M2T transformations using template languages. Section 3 lists existing work available on the subject. Section 4 describes the design pattern in a format that is similar to those that are used in design pattern catalogs. Section 5 concludes.

2 Background

M2T transformations are often used for generating source code, documentation or other artefacts in order to speed up software development. Models can be easily produced and validated by the proper tooling so we can make sure they convey the correct information. By using models and code generation techniques, the resulting source code can be produced more quickly. Besides, if they are validated and the code generator is correct, we can assure that the generated code is correct as well. When using *Domain-Specific Modeling (DSM)* and model processing [1-2], these benefits apply at a much higher extent. DSMs raise the abstraction and allow for expressing the problem from a more human-friendly viewpoint that does not require thinking in programming notions. This makes the process less error-prone.

Because of the above reasons, the code generator is an especially important component in model processing. To facilitate the development of code generators, so called template languages have been developed. Generating code from conventional programming languages is difficult because substitution and print instructions are intermixed with literal fragments of the output, thus it becomes hard to read. Template languages reverse the logic: everything written in the template goes to the output by default, only value substitutions and conditional instructions or loops must be written with special markup. These template languages highly simplify the development of code generators and make the generated code easier to read and write. However, as the complexity of the model grows, templates also become longer and more complex and these advantages can be only achieved at a limited extent. From different model objects, usually different kinds of code fragments are generated and this requires branching instructions in the template. If there is a high number of them, the template becomes hard to read and maintain.

Despite that the explanation above mentions mostly code generation, other kinds of M2T transformations, like generating reports, documentation etc. are very similar in nature and the pattern is also applicable to them. For the sake of simplicity, throughout the paper we will simply refer to the M2T transformations as code generators.

3 Related work

The first well-known work that proposed the reuse of working solutions to common software engineering problems and their description in design pattern catalogs was the one published by Gamma et al. [3]. This work was followed by the *Pattern-Oriented Software Architecture (POSA)* series [4,5,6,7,8]. Apart from these general object-oriented design patterns, some more specialized patterns have also been described. In the field of *Domain-Specific Languages (DSLs)*, [1] provides a pattern catalog, covering several different aspects of DSLs and code generation. This is a rich source of information but it has a more general view than this paper and does not include the pattern described herein. Apart from this, [9] provides some practical uses of general object-oriented design patterns in recursive descent parsers and [10] describes how a parser generator uses object-oriented design patterns. These are specific uses of general design

patterns and these papers do not include more specialized patterns specific to DSLs and code generation. A pattern catalog [11] of architectural design patterns that can be used in language parsers has also been published. This is relevant for implementing DSLs.

Beside the movement of collecting solutions of common problems in design pattern catalogs, Yu and Mylopoulos emphasized [12] that research of software engineering had focused more on the what and the how rather than on the why. Their contribution justifies the need for more work that deals with understanding the requirements. There are also publications that collect the intents of using specific software techniques in so called intent catalogs. These are similar in nature to design patterns but they describe common motivations behind applying a specific solution. The intents behind DSLs have been described in [13]. Amrani et al. has published an intent catalog behind using model transformations [14].

There are several existing template engines that allow for the decomposition of templates into smaller units. These make it possible to organize template code into separate methods and files. By using these tools, each model class can have its own template associated and template code can be further cut down to methods that generate a specific feature from the model class. For example, such template implementations are *Xtend* [15] and *Microsoft T4* [16]. The pattern presented in this paper provides a method for using these tools efficiently.

The technique of incremental model transformation [17] is related to this work in that it also deals with changes in the code generation process. However, this approach handles changes in the input model and is able to update parts of the generated model based on the changes in the input model. By not having to rerun the entire transformation, it saves computational time. In contrast, the design pattern presented herein facilitates the evolution of the M2T transformation itself. As the tool evolves and more features are supported by the code generator, the templates also becoming more complex. The proposed solution decomposes the templates into highly cohesive, flexible units to facilitate changes and extension. So the two techniques address different issues and are not mutually exclusive.

4 The Polymorphic Templates Design Pattern

This section describes the design pattern in catalog format similar to what is used in the *POSA* series. Namely, the following sections are applied:

- *Example:* a concrete use case in which the pattern has been applied.
- *Context:* the context in which the design pattern is applicable.
- *Problem:* the challenges that suggest the application of the pattern.
- *Solution:* the way how the pattern solves or mitigates the problems.
- *Structure:* the main participants and their relationships and responsibilities in the pattern.
- *Dynamics:* the interaction of the participants of the pattern.
- *Implementation:* techniques and considerations for implementing the pattern.
- *Consequences:* advantages and disadvantages that the application of the pattern implies.

- *Example Resolved:* the short description of how the initially presented example has been resolved by using the pattern.

The *Known Uses* and the *See Also* sections are omitted due to lack of space and related patterns.

Example

The *ProtoKit* tool [18] is a DSL and an accompanying code generator for describing the message structure of application-level binary network protocols. Object-oriented general purpose programming languages (GPLs) can represent messages as classes, being the member variables the different fields encompassed in the message. However, several features of these protocols are difficult to support in this way, such as bitfields, encoded fields or length fields of variable-length fields. *ProtoKit* generates the classes, member variables and accessors with the boilerplate code to support the above mentioned features. In the generated code, different fields of the message will result in different variable definitions, initialization code snippets and accessor methods. Because of this, the template of the generated code contains several loops that iterate over the fields and each iteration includes several *instanceof* checks. Because of the looping and branching markup in the template, the actual output is hard to read among the lines. Besides, the template is rapidly growing as new features and model classes are added. Parts of the template are not decomposed according to what feature they generate or what model object they process. This makes it difficult to locate and modify the generated code of a specific feature.

Context

Model-to-text transformations that use templates to produce textual output and have a type hierarchy in the input model.

Problem

When complex models are processed from template languages a number of problems arise:

- *High complexity.* The template class can be decomposed into several methods that are responsible for generating different features but the template that processes the model remains highly complex.
- *Lack of encapsulation and separation of concerns.* Code fragments for generating different features from the same model class are separated by branching instructions and thus are scattered through the template. Logically coherent code fragments are not encapsulated into highly cohesive classes.
- *Poor readability.* Because the output code is intermixed with conditions and branching instructions that generate the proper code fragment from each model object that

is traversed, the main goal of using a template language – good readability – does not apply.
- *Poor maintainability and extensibility.* Because of the lack of encapsulation and the branching instructions that add syntactic noise, several isolated parts of the generator must be modified in order to modify the behavior. Similarly, extension requires adding new code fragments to several places inside the same template.

Solution

Decompose templates on a per model class and per feature basis. The code fragment generated for a specific feature and from a specific model class will be encapsulated in its own template. The generator accesses templates via their common interfaces and does not know about concrete template types. Determining the template to call for a specific feature is the responsibility of the model object being traversed.

Structure

A possible structure of the pattern is depicted in Figure 1. To keep the diagram comprehensible, only one feature, *feature A* is depicted. Of course, the pattern supports multiple features by having multiple feature hierarchies. The pattern has the following participants:

- *ModelClass:* an abstract type of the model elements processed by the generator.
- *ConcreteModelClass1* and *ConcreteModelClass2:* concrete types with different semantics that are instantiated in the model. They usually result in different output.
- *FeatureATemplate:* an interface for the polymorphic templates that generate *feature A* for concrete instances of *ModelClass.*
- *ConcreteFeatureATemplate1* and *ConcreteFeatureATemplate2:* concrete templates that generate "*feature A*" for *ConcreteModelClass1* and *ConcreteModelClass2,* respectively.
- *Generator:* the entry point of the code generator. This component traverses concrete instances of *ModelClass* in the model and calls the corresponding templates.
- *Application:* the main application that obtains the model (depicted as a set of aggregated *ModelClass* instances) and calls the code generator.

Dynamics

The Generator component has knowledge of what features must be generated and in what order. This is specific to the domain. Generating features involves the traversal of model objects in the input model. When a specific model object is visited, the Generator first calls the *getATemplate()* method on the model object to obtain an instance of the template to use. Then the Generator calls the *generateA()* method on the template instance obtained from the visited model object. This is depicted in Figure 2.

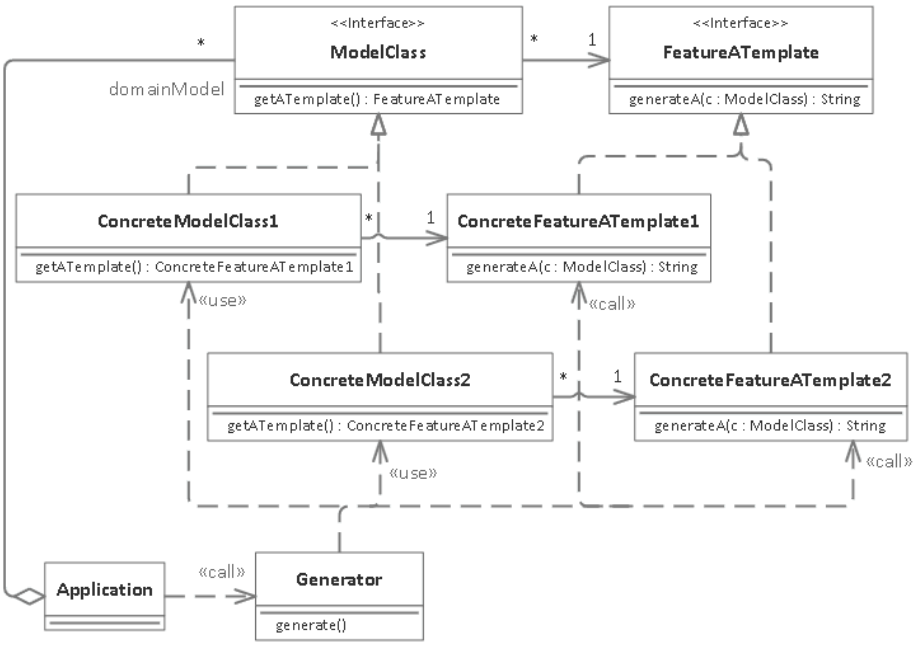

Fig. 1. A possible structure of the participants in the *Polymorphic Templates* pattern

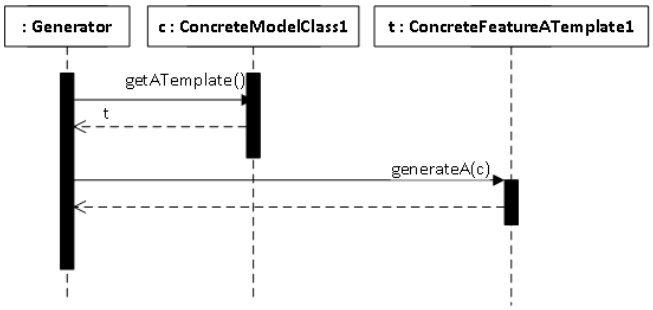

Fig. 2. The interaction of the participants in the *Polymorphic Templates* pattern

Implementation

The following techniques should be considered for the implementation of the pattern:

- Model classes may return template references in three different forms:

  (a) *As object references.* This is the most object-oriented, the most efficient and the safest option. The code generator directly retrieves a reference and uses it to call

the template. To apply this solution, the modeling framework must support references to classes that are not part of the model. Several modeling frameworks support this by importing external data types.

(b) *As the class name.* The class name is returned as a string to the code generator and before calling the template, the code generator must instantiate the template class using reflection. Reflection has some performance hit and the class name is easy to mistype. However, this is a viable option if the modeling framework does not support external data types.

(c) *As an identifier.* An arbitrary identifier that is used to obtain a reference for the concrete template class. For example, it may be a key for a hash table. This option eliminates the performance hit of reflection but the validity of the identifiers still must be guaranteed by the implementor. This solution requires an extra effort to implement the lookup mechanism.

- Deciding what is an independent feature is a crucial point in applying this patterns. This determines the number of templates and how cohesive individual templates will be. This latter greatly affects flexibility and extensibility. It does not only include identifying features per a single class hierarchy but deciding on cases like whether a single template will be provided for aggregating elements or it will be chunked down to separate ones, one per each aggregate.
- If the template references are obtained by instance methods, template inheritance can be leveraged. In case a parent and a child model class generate the same code for a specific feature, the template can be inherited. It is not necessary to define a new template for the child, nor to repeat the template reference.
- The pattern can be implemented through model refinement if the modeling environment supports it. In this scenario there are two layers of models. The lower layer contains information that is strictly the model without template associations. The upper layer refines the lower layer, adding references to templates. If there are several target languages, it is a viable solution to define a separate upper layer for each of them.

Consequences

The pattern achieves the following advantages:

- *Reduced complexity of the templates.* The complexity of templates is reduced by decomposing long and complex ones that contain conditions and branching markup into several shorter “flat” templates.
- *Separation of concerns in the code generator.* Each template is responsible for the generation of a specific feature using a specific model class.
- *Improved readability.* The eliminated branching instructions make templates more readable. Code is not intermixed with control statements therefore the actual code to be emitted is easier to understand. A good abstraction of features and template hierarchy helps to factor templates in a way that each of them contains an intuitively comprehensible unit of code fragment. This further improves readability.

- *Improved extensibility.* The implementation of a concrete feature regarding a concrete model element is encapsulated into a concrete template class. This, combined with an intuitive and consistent naming convention, makes it easy to locate the code that must be modified. As a result, modifications of a specific feature are constrained to a small set of templates or even a single template if it only affects a single concrete model class. The generator itself or templates of other features or unaffected concrete model classes never need modifications. Extensions are similarly straightforward. The only necessary steps are creating the templates for features of the newly added concrete model class and associating them with the new class. Since the generator only has knowledge of the common interface and calls the concrete templates through polymorphism, it is not necessary to modify the generator in any way.

The application of the pattern also has some disadvantages:

- *Mixing model and implementation details.* Although the pattern achieves good separation of concerns in the code generator, it mixes some implementation details with the model. The pattern associates templates with model classes despite that the latter should be part of the code generator itself since it conveys implementation details for the generator. On the other hand, it can be argued that the features generated from model classes has to do with their behavior, which in turn, fits into the model. Developers that apply the pattern should consider whether it is a problem for them that such details are stored in the model. If there is no benefit in separating them from the model, this issue should not be blindly considered as a severe problem just because best practices warrant of separating model and implementation. However, if there is a high number of features or there are several supported target languages, models may become flooded by template references.
- *Increased number of template classes.* Since templates are decomposed on a per feature and a per concrete model class basis, their number increases significantly. This is a direct consequence of decomposition so it is not considered as a real disadvantage. Besides, the number of classes depends on the choice of feature abstraction and the use of template inheritance, therefore it can be slightly adjusted by choosing the right abstractions.
- *Difficult to deal with cross-cutting features.* It is not trivial how to apply the pattern with cross-cutting features that are not associated to a single model class but to several ones. If there is an aggregate that references the involved model classes, it can be considered to associate the features to that model class.

Example Resolved

In the *ProtoKit*[1] tool, several features have been identified: (1) variable definition, (2) initialization code, (3) accessor methods, (4) equals expression to compare the generated variables, (5) hashcode expression to generate a hash for the variable and (5) clone code for the variable. For the implementation of the tool, the *Eclipse Modeling Framework (EMF)* [19] has been used. EMF leverages round-trip code generation and allows

[1] Source code available at https://github.com/gaborbsd/ProtoKit.

for defining methods on model classes by writing their Java code. The abstract *Field* model class is the superclass of all fields that can be used in a message definition and it defines the methods that return an instance of the template to use for the concrete *Field* instance. The templates are implementations of the *FieldGenerator* interface. It is imported to the EMF model as a data type so that the methods can be modelled. The interface only defines a *generate()* method that takes the a *Field* instance as an argument. This generates the code fragment of a specific feature from the specified *Field.* The implementations of the interface implement this method and do not store state so they use the *Singleton* [3] pattern to facilitate the reuse of instances. The templates have been decomposed on a per field per feature basis as the *Polymorphic Templates* pattern suggests. Not all of the Field types require their own template, some templates are reused. For example, Fields that generate a primitive Java variable all share an empty template for the cloning feature since Java's *clone()* method inherited from *Object* takes care of them. Referential variables all cloned in the same way, so they also share a template.

The application of the design pattern has highly improved the readability and the flexibility of the tool. It became easier to add new *Field* types and to modify existing functionality. The increased number of the templates do not cause a problem and they have been grouped to Java packages based on the feature they generate. Because of EMF allowing method definitions on model classes and the moderate number of features on the type hierarchy of *Fields,* mixing models and implementation did not mean any disadvantage either. In the tool, there were no cross-cutting features so the pattern was easy to apply.

5 Conclusion

The paper has presented a novel design pattern for implementing agile M2T transformations. This solution has been identified in our DSL-based tools that leverage code generation to help software development. It has been chosen to publish this solution as a design pattern to facilitate its reuse. The catalog format enables developers to easily understand the context of the application, the problems that arise in this context and how the application of the design pattern addresses these issues. Implementation ideas are also provided. These help developers to decide, which variant fits better their needs. A real-life application of the pattern will be published in our future papers.

We believe that the *Polymorphic Templates* design pattern will be of great use for other software developers who use M2T transformations. It is a potential tool for creating agile code generators and thus is highly demanded in nowadays' software environments.

Acknowledgments. This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013) organized by VIKING Zrt. Balatonfüred. This work was partially supported by the Hungarian Government, managed by the National Development

Agency, and financed by the Research and Technology Innovation Fund (grant no.: KMR_12-1-2012-0441).

References

1. Fowler, M.: Domain-Specific Languages, Addison-Wesley (2010)
2. Kelly, S., Tolvanen, J.: Domain-Specific Modeling: Enabling Full Code Generation, Wiley - IEEE Computer Society Publications (2008)
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley (1995)
4. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture Volume 1: A System of Patterns, Wiley (1996)
5. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects, John Wiley & Sons (2000)
6. Kircher, M., Jain, P.: Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management, Wiley (2004)
7. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing, Wiley (2007)
8. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages, Wiley (2007)
9. Nguyen, D., Ricken, M., Wong, S.: Design Patterns for Parsing, In: 36th SIGCSE Technical Symposium on Computer Science Education, pp. 477–48, ACM, New York (2005)
10. Schreiner, A.T., Heliotis, J.E.: Design Patterns in Parsing, In: 10th IEEE International Symposium on High Performance Distributed Computing, pp. 181–184, IEEE Press, New York (2001)
11. Kövesdán, G., Asztalos, M., Lengyel, L.: Architectural Design Patterns for Language Parsers, Acta Polytechnica Hungarica, vol. 11, no. 5, pp. 39–57 (2014)
12. Yu, E.S.K., Mylopoulos, J.: Understanding 'why' in software process modelling, analysis, and design, In: Proceedings of 16th International Conference on Software Engineering, pp. 159–168., IEEE Computer Society Press (1994)
13. Kövesdán, G., Asztalos, M., Lengyel, L.: A classification of domain-specific language intents, International Journal of Modeling and Optimization, vol. 1, no. 4, pp. 67–73 (2014)
14. Amrani, M., Dingel, J., Lambers, L., Lúcio, L., Salay, R., Selim, G., Syriani, E., Wimmer, M.: Towards a model transformation intent catalog, In: Proceedings of the First Workshop on the Analysis of Model Transformations, pp. 3-8, ACM (2012)
15. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, Packt Publishing (2013)
16. Vogel, P.: Practical Code Generation in .NET, Addison-Wesley (2010)
17. Hearnden, D., Lawley, M., Raymond, K.: Incremental Model Transformation for the Evolution of Model-Driven Systems. In: Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 4199, pp. 321-335 (2006)
18. Kövesdán, G., Asztalos, M., Lengyel, L.: Modeling Cloud Messaging with a Domain-Specific Modeling Language, In: CloudMDE, A Workshop to Explore Combining Model-Driven Engineering and Cloud Computing. In conjunction with MoDELS 2014. In press.
19. Steinberg D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd Edition, Addison-Wesley Professional (2008)