Aggregate Callback
A Design Pattern for Flexible and Robust Runtime Model Building

Gábor Kövesdán, Márk Asztalos and László Lengyel
Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Budapest, Hungary

Keywords: Modeling, Domain-Specific Modeling, Model Transformation, Code Generation, Design Pattern, Agility.

Abstract: In modern software engineering environments, tools that use Domain-Specific Languages (DSLs) are often applied. The usual workflow of such tools is that the textual input written in the DSL is parsed and a semantic model is instantiated. This model is later passed to another software component that processes it, e.g. a model transformation, a code generator or a simulator. Building the semantic model inside the parser is often a complex task. The model must be built in such a way that the constraints of the problem domain are enforced so that the consistency of the output is guaranteed. This paper presents a design pattern, referred as *Aggregate Callback* that supports enforcing constraints in the model and thus helps creating correct models. We have found that the *Aggregate Callback* pattern is useful for tool developers that build models in their applications.

1 INTRODUCTION

Model-Driven Development (MDD) (Brambilla, Cabot and Wimmer, 2012) relies on modeling the problem and using that model through an arbitrary number of refinement steps, called model transformations (Syriani and Vangheluwe, 2009). This model is often used later for code generation. It facilitates and speeds up software development since the model does not have to include all of the implementation details that are added later by the refinement steps. This approach provides several advantages, e.g. improves product quality and reusability. As a consequence, developers can focus on the real problem instead of monotonous coding. However, the resulting software can only be expected to be correct if the model is constructed in the way as expected by the code generator. To ensure this, modelers first define the metamodel of the models (Kühne, 2006), also called abstract syntax. The metamodel is a type that defines the possible structure of model instances that are created from the metamodel. Apart from this, other requirements that are not captured in the structure can be added through constraints. Before the model is processed, a validation step (Brambilla et al., 2012) is performed, which checks whether the model is valid, that is, if it conforms to the metamodel and to the constraints.

However, validation is purely a passive check. If the model does not conform to the metamodel or the constraints, an error is emitted with the location of the problem and the developers have to manually fix the model. In *Domain-Specific Modeling (DSM)* (Fowler 2010) (Kelly and Tolvanen, 2008) tools, validation is not sufficient for building a robust modeling tool. When the model is built in the program, for example as a result of traversing the parse tree of an input script written in a *Domain-Specific Language (DSL)* (Fowler, 2010) (Kelly and Tolvanen, 2008), the program must ensure that the created model is valid. DSLs raise the abstraction level at which problems are described. This means that DSM tools encompass domain knowledge. Some interrelations and constraints among model objects, such as dependency, exclusion, calculated attribute, etc. can be inferred from the nature of the problem domain. These details do not have to be described by the user of the DSL but it is the responsibility of the tool to handle such cases correctly. If the DSL required developers to describe these inferable details, the language would not be so concise and we would lose its common advantages, such as higher abstraction level and quick development. So in these cases, the tool is responsible for ensuring some of the constraints of the problem domain correctly. Validation can point out inconsistent models that are the result of a software bug. However, validation itself does not facilitate organizing well the source code of the tool so it is desired to find a way that helps ensuring that the tool produces valid models.

Kövesdán G., Asztalos M. and Lengyel L..
Aggregate Callback - A Design Pattern for Flexible and Robust Runtime Model Building.
DOI: 10.5220/0005197901490156
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2015), pages 149-156
ISBN: 978-989-758-083-3

In this paper, we present a design pattern, called *Aggregate Callback,* which helps tool developers to structure the code in a way that makes the model building logic more flexible and robust. We believe that this design pattern will greatly help developers of modeling tools in designing agile tools that are easier to maintain and extend. We have used this design solution in earlier tools but the detailed description of this approach as a reusable design pattern is a new contribution in this paper.

The rest of this paper is organized as follows. Section 2 lists existing work available about the subject. Section 3 describes the design pattern in a format that is similar to those that are used in design pattern catalogs. Section 4 concludes the paper.

2 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 catalog was the one published by Gamma et al. (Gamma et al., 1995). This work was followed by the *Pattern-Oriented Software Architecture (POSA)* series (Buschmann et al., 1996) (Schmidt et al. 2000) (Kircher and Jain, 2004) (Buschmann, Henney and Schmidt, 2007a) (Buschmann, Henney and Schmidt, 2007b). Apart from the design patterns applicable in general software engineering problems that were described in these books, more specific design patterns have also been published. In the field of DSLs, (Fowler, 2010) 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, (Nguyen, Ricken and Wong, 2005) provides some practical uses of general object-oriented design patterns in recursive descent parsers and (Schreiner and Heliotis, 2001) 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 modeling. A paper have been published with a pattern catalog (Kövesdán, Asztalos and Lengyel, 2014a) of architectural design patterns that can be used in language parsers. Another paper (Kövesdán, Asztalos and Lengyel, 2014b) introduces *Polymorphic Templates*, a design pattern that provides a solution for implementing flexible code generators. An additional case study (Kövesdán, Asztalos and Lengyel, 2014c) briefly describes the use of the *Aggregate Callback* and the *Polymorphic Templates* design patterns. However, the *Aggregate Callback* design pattern is only shortly outlined in this earlier paper. It has not been elaborated at the level of detail as done herein. Apart from the works cited above, we have not found other works that provide more specialized design patterns that apply to modeling.

The design pattern is useful on its own but it is also a part of a longer work that the authors have been doing. A more extensive method is being elaborated that helps the development of code generation tools supported by DSLs.

3 THE AGGREGATE CALLBACK 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.
- *See Also:* references to related design patterns.

The *Known Uses* section is omitted. Describing more known uses is out of scope of this paper.

3.1 Example

Applications written for the Android platform have several different component types and artefacts, namely *Activities*, *Services*, *Content Providers*, *Intents*, *Intent Filters* etc. Some of them have more specific subtypes, such as *IntentService.* In the metamodel of our component modeling tool, they are organized into a class hierarchy. The *AndroidApplication* model class aggregates an arbitrary number of *Components,* regardless of their concrete type.

Certain component types imply some constraints that must be enforced in order to have a consistent model that represents a working Android application. For example, if a *GCMBroadcastReceiver* is added to the application to handle *Google Cloud Messaging (GCM)* (Google, n.d.) notifications, the main *Activity* of the application must initialize the GCM service and the *Manifest* file must declare some permissions and metadata related to message handling. Validation can detect if the developer did not specify a *GCMActivity* but the permissions and the metadata are not described in the input text by design. If they were also described, the DSL would not be as concise as it should be. These details can be inferred so they must be added to the model when a GCM-related *Component* is added to *AndroidApplication.*It seems logical to handle this in the *AndroidApplication* class. However, the code of the class would include too much information about component types in this way, which limits the flexibility because of the lack of the separation of concerns. Extending the solution with support for other component types would be difficult since the aggregate class would require modifications to enforce new constraints.

3.2 Context

The pattern is used in modeling tools that build models at runtime and must enforce some constraints among model elements as the model elements are aggregated in the model.

3.3 Problem

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

- *Limited Functionality.* If the constraints are only validated and are not enforced, constraint violations can only be detected but it is not possible to enforce constraints. This requires the DSL that is used for modeling to describe each detail of the model, even those that could have been inferred by the specific characteristics of the problem domain. Easy to use DSLs should achieve conciseness by not describing inferable details of the model.
- *High Complexity.* If dependency handling and other constraints are implemented in a centralized way – either in the application or in the aggregate model object – the complexity becomes high. The logic will be implemented as a big piece of code without proper decomposition.
- *Lack of Encapsulation and Separation of Concerns.* Such implementation does not encapsulate the code that deals with constraints based on what model class they belong to.
- *Poor Readability.* Because of the lack of encapsulation, the overall effect of the code is hard to understand.
- *Poor Extensibility.* Because of the lack of encapsulation, several isolated parts of the code must be modified if a new model class is added to the metamodel. Not only the model class must be implemented, but the centralized constraint handling logic must also be updated on a regular basis. This limits extensibility.
- *Error-prone Application.* The former problems lead to an error-prone application because it is easy to make human errors.

3.4 Solution

Make enforcing constraints a responsibility of the aggregated model objects. When a new model object is added to the model, a callback method is called on the object. Through the callback method, the object receives the reference of the model and by using it, will be able to walk along other objects already aggregated into the model and apply the changes that are necessary to enforce constraints.

3.5 Structure

A possible structure of the pattern is depicted in Figure 1. The nomenclature reflects a dependency constraint but the pattern can be used for other kinds of constraints as well. The pattern has the following elements:

- *Application:* the main application logic that creates the *Model* and its aggregated instances of *ModelClass.*
- *Model:* the model itself that aggregates several instances of *ModelClass.*
- *ModelClass:* abstract class, whose instances may be added to the *Model.*
- *DependencyModelClass:* a concrete subclass of *ModelClass* that may be used as a dependency for other instances of *ModelClasses.*
- *DependentModelClass:* a concrete subclass of *ModelClass,* whose instances depend on instances of *DependencyModelClass.*

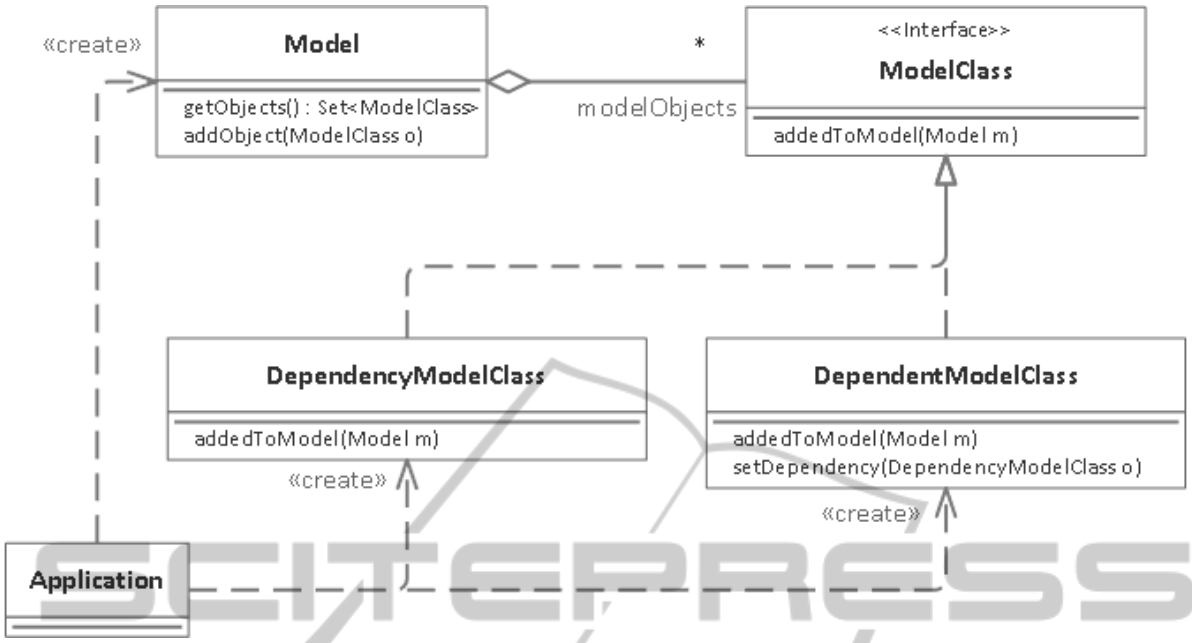

Figure 1: A possible structure of the participants in the *Aggregate Callback* pattern.

3.6 Dynamics

The dynamics are demonstrated on enforcing a dependency relation. The *Application* creates the *Model* and instantiates the objects that will be added to the model. Usually, standalone objects are added first that do not enforce any dependency relations. Later, further objects may be added, that depend on other objects added earlier. If the order in which objects are added to the model is not guaranteed, the constraint must be checked and enforced in all objects that are affected by the particular constraint. If objects are added in a fixed order, it is sufficient to enforce the constraint.

Objects are added with the *addObject()* method of *Model.* After adding an object, the *Model* calls back its *addedToModel()* method, passing itself as a parameter. Concrete *ModelClasses* use this method to implement the necessary logic for enforcing constraints, such as dependency relations. It is possible to query other objects from the *Model* by calling its *getObjects()* method, walking on the other objects and modifying them. When a *DependentModelClass* is added, it can traverse *ModelClasses* and check if a proper *DependencyModelClass* exists in the *Model.* Such a scenario is depicted in the sequence diagram of Figure 2. The reference to the dependency class is stored by calling the *setDependency()* method.

3.7 Implementation

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

- The pattern can be applied to the complete model or any other subset that contains an aggregate object and associated aggregated objects.
- In the description of the pattern, aggregation is mentioned, however, the pattern is also applicable to associations. The concept that is emphasized with the name of the pattern is that it works among objects that together form the model or a logical set of objects.
- Constraints may be of various types: dependency, exclusion, calculated value etc.
- From the callback method, it is possible to simply modify attributes of other model objects or performing more complex operations as well, such as, adding or removing object or associations.
- There may be constraints that involve more than two model objects. The developer should carefully consider in which model class to implement the constraint enforcement. If it is guaranteed that they are added to the model in a specific order, enforcing constraints in the last model class is a reasonable solution

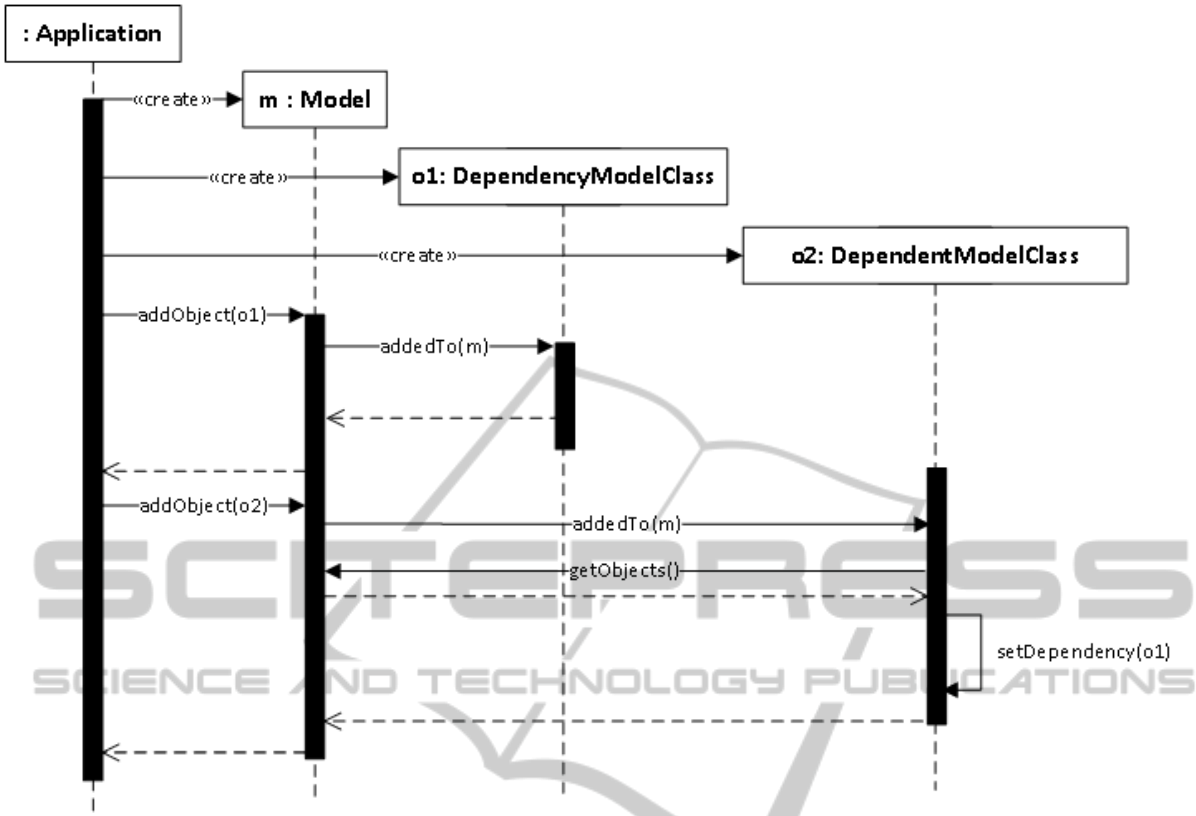

Figure 2: The interaction of the participants in the *Aggregate Callback* pattern.

Otherwise, it is recommended to implement constraint check and enforcement in all participants. This may be done with inheritance to avoid redundancy in the code. This solution has a performance hit since the constraint check is performed several times. However, if the model is used in a code generator, this is not a problem since it does not affect the performance of the final (generated or partly generated) software product.

3.8 Consequences

The pattern achieves the following advantages:

- *Separation of Concerns in the Modeling Tool.* Each model class is responsible for checking and enforcing the related constraints.
- *Easy Maintainability and Extensibility.* When adding model classes, new and related constraints can be implemented in the callback method of the new model class. Optionally, some old model classes that also participate in new constraints will need to be updated. Apart from this, the code does not require modifications.
- *Robust Application.* Structuring the code in this way helps to systematically implement constraint enforcement and ensure that the tool produces a valid model.

The application of the pattern also has a disadvantage:

- *Hard to See all the Constraints and the Overall Effects of Callbacks.* The constraint enforcing logics are decomposed based on what model class they belong to. From a responsibility point of view, this achieves good separation of concerns since a specific logic is encapsulated into the model class, which is involved in the specific constraints. However, if we want to review constraint enforcing as a whole, we face difficulties since the code is scattered across model classes. This is a direct consequence of the application of the pattern since its main idea was to decompose constraint enforcement logic. Therefore, this is not a serious disadvantage. A possible way to mitigate this problem is to factor out different constraint checks into methods of a single class and calling these from the callbacks. In this way, all of the advantages of the design pattern apply and the constraint checking code remains easier to understand.

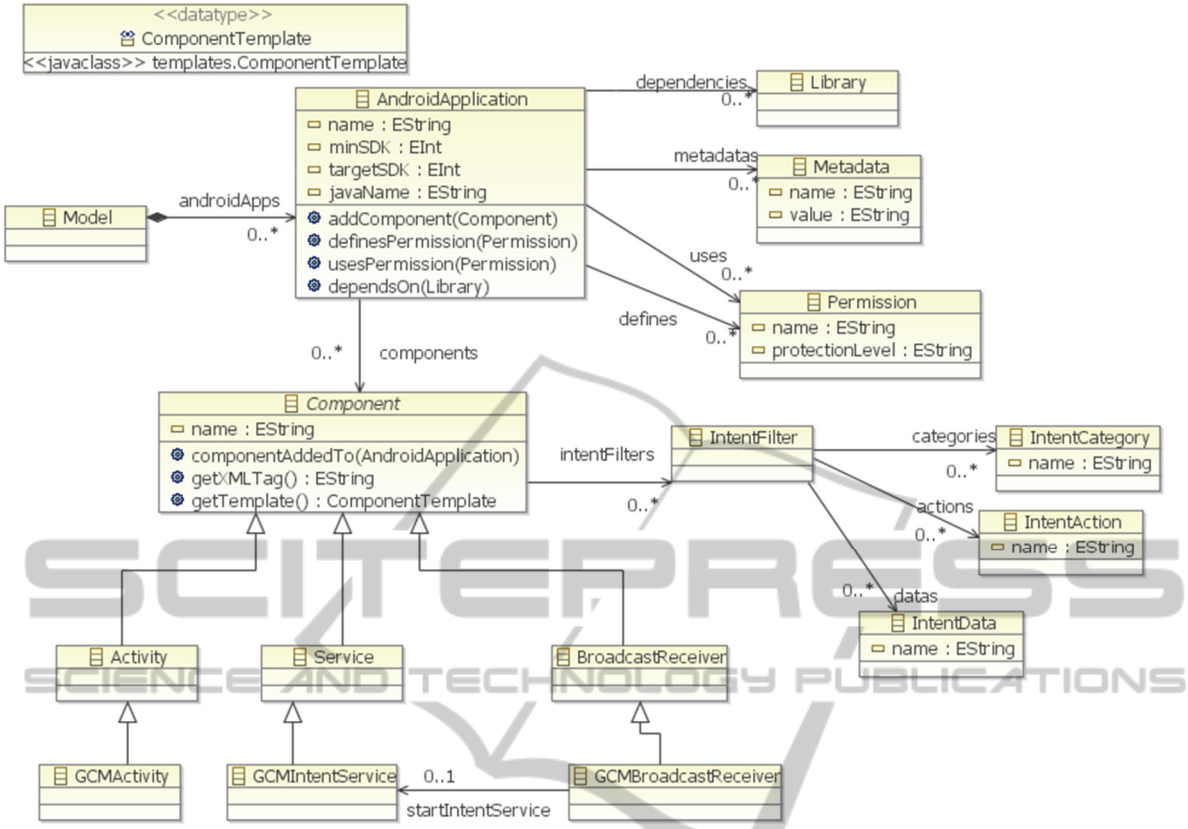

Figure 3: The metamodel used in the AndroidModeler tool.

3.9 Example Resolved

In the *AndroidModeler* tool (Kövesdán, Asztalos and Lengyel, 2014c), the pattern has been successfully applied. The metamodel has been created with the *Eclipse Modeling Framework (EMF)* (Steinberg et al, 2008). The EMF framework leverages round-trip code generation and allows for defining methods on model classes in Java. This made it easy to implement the callback methods on *Components.* The metamodel used in the tool is depicted in Figure 3. The *AndroidApplication* model class defines an *addComponent()* method that can be used to add *Components* to the application model. This method stores the new *Component* and then calls back the *componentAddedTo()* method on the *Component,* passing the *this* reference as a parameter. From this method, concrete subtypes of the abstract *Component* class can traverse and modify other elements of the model, thus enforcing the constraints. When an Android application uses GCM, some constraints must be enforced. This could be done from the callback of any of the three GCM-related components, namely *GCMActivity, GCMBroadcastReceiver* or *GCMIntentService.* It was chosen to enforce these constraints in the *GCMBroadcastReceiver* component, since it is the main component that initializes GCM and it is obligatory in each GCM-enabled application. The callback first creates a *Metadata* model object with the name *com.google.android.gms.version* and the value as the current GCM version number and adds it to the application. This is required by the GCM framework. After this, several types of *Permissions* are created. Most of these are declared as used by the application but there is also a *Permission* defined by the application. This serves for receiving GCM messages. Finally, an *IntentFilter* is set up to route GCM messages properly.

As outlined earlier, details such as the metadata on the GMS version or the different kinds of permissions are not part of the textual description. These are details that can be inferred and are not relevant at the level of the component modeling. Therefore, these should be added automatically in the tool. Validation can be used in a later phase to check whether these constraints hold but such validation only works as a test for the software. It does not validate input coming from the user. Provided that the tool works correctly, it will always pass. To highlight the difference between the objectives of the *Aggregate Callback* des

ign pattern and validation, we will explain a different aspect of the tool. The components that are used to handle GCM messaging are specified in the DSL. A *GCMActivity* is always obligatory. It is the main component that registers the application to the servers of Google. Apart from this, a *GCMBroadcastReceiver* is also necessary to receive incoming cloud messages. It is a common practice to dispatch the handling of *Intents* in an *IntentService.* To facilitate such a solution, a *GCMIntentService* component type has been also introduced. This component is optional in GCM-enabled applications. If added to the application, such a *BroadcastReceiver* is generated that dispatches the event handling to the *IntentService.* Since these components are configured in the textual description, the generated components depend on the intentions of the user. Therefore, in this case, validation should be used to check whether the specified component structure is valid.

In the *AndroidModeler* tool, the application of the pattern achieved good separation of concerns. The code that handles these constraints was encapsulated into the *GCMActivity* class. Other pieces of the code are not affected by the constraints and the *AndroidApplication* class is not polluted by code that logically belongs to the constraints of individual components. The tool is still in an early phase but it will be easy to add further specialized component types because the existing code will not require modifications. Besides, the constraints on these specific model classes will surely hold in the resulting model since the tool is written with this in mind. The disadvantage of the pattern was that it is not straightforward to find these pieces of code. It may take an effort for developers unfamiliar with the code to understand the inner working of the tool. The empty callback method is defined on the abstract type of the aggregated components, that is, *Component,* but concrete implementations may freely override it. This requires the developer to read all the descendants of *Component* in order to understand the overall effect of the code. Furthermore, as outlined in Section 3.7, constraints that involve several *Components,* can be enforced in any of them. This makes it even more difficult to understand the code. In the *AndroidModeler* tool, it did not cause problems because the tool aims to only generate a skeleton, so it is an inherently simple application. In more complex code generator tools, this may be a more serious problem.

3.10 See Also

The *Polymorphic Templates* design pattern (Kövesdán et al., 2014b) is related that it is a specific design pattern for code generators. It reduces the complexity of long templates by decomposing them on a per feature per model class basis. It was also applied in the *AndroidModeler* tool on the templates of the *Components.*

4 CONCLUSIONS

The paper has presented a novel design pattern for building models in a more robust and flexible way. This solution has been identified in our DSL-based tools that build models from textual input. 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. An earlier paper (Kövesdán, Asztalos and Lengyel, 2014c) includes a case study in which the pattern is applied in a real application.

We believe that the conscious application of the *Aggregate Callback* design pattern will greatly help the development of DSL-based applications. It is a potential tool for creating agile modeling tools and thus is highly demanded in modern software environments. The design pattern is applicable itself but it is also part of a method being elaborated on building code generator tools that use DSLs as input. Future work on the subject includes an extensive description of the complete method.

ACKNOWLEDGEMENTS

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

Brambilla, M., Cabot, J., Wimmer, M., 2012. *Model-Driven Software Engineering in Practice.* Morgan and Claypool.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns.* Wiley.

Buschmann, F., Henney, K., Schmidt, D.C., 2007a. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing.* Wiley.

Buschmann, F., Henney, K., Schmidt, D.C., 2007b. *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages.* Wiley.

Fowler, M., 2010. *Domain-Specific Languages.* Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Google, n.d., Google Cloud Messaging for Android. Available from: *http://developer.android.com/google/gcm/index.html.*

Kelly, S., Tolvanen, J., 2008. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley - IEEE Computer Society Publications.

Kircher, M., Jain, P., 2004. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management.* Wiley.

Kövesdán, G., Asztalos, M., Lengyel, L., 2014a. Architectural Design Patterns for Language Parsers, *Acta Polytechnica Hungarica,* vol. 11, no. 5, pp. 39–57.

Kövesdán, G., Asztalos, M., Lengyel, L., 2014b. Polymorphic Templates. In: *XM 2014 Extreme Modeling Workshop, In conjunction with MODELS 2014. In press.*

Kövesdán, G., Asztalos, M., Lengyel, L., 2014c. Fast Android Application Development with Component Modeling. In: *5th Conference on Cognitive Infocommunications*. *Submitted for publication.*

Kühne, T., 2006. *Matters of (Meta-) Modeling.* Journal on Software and Systems Modeling, vol. 5, no. 4, pp. 369-385.

Nguyen, D., Ricken, M., Wong, S., 2005. Design Patterns for Parsing, In: *36th SIGCSE Technical Symposium on Computer Science Education,* pp. 477–48. ACM.

Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F., 2000. *Pattern-oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects.* John Wiley and Sons.

Schreiner, A.T., Heliotis, J.E., 2001. Design Patterns in Parsing, In: *10th IEEE International Symposium on High Performance Distributed Computing,* pp. 181–184, IEEE Press.

Steinberg D., Budinsky, F., Paternostro, M., Merks, E., 2008. *EMF: Eclipse Modeling Framework.* 2nd Edition, Addison-Wesley Professional.

Syriani, E., Vangheluwe, H., 2009, *Matters of model transformation.* School of Computer Science, McGill University, SOCS-TR-2009.2.