RWTH Aachen University

Software Engineering Group

FH Aachen, Campus Jülich

09 Medical Engineering and Technomathematics

Patterns in generative software development

Seminar thesis

Presented by

Ali, Daoud

MatrNr: 3278373

1st examiner: Prof. Dr. rer. nat. Bado Kraft

2nd examiner: M.Sc. Alexander Hellwig

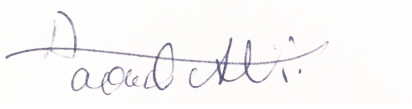
Aachen, December , 2022

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema
Patterns in generative software development
selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.
Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Name: **Daoud Ali**
Aachen, den 31.12.2022

Unterschrift des Studenten



Abstract

The principle of software generation has been getting more attention in recent years and has become an important paradigm in software engineering. Generative software development simply put means generating code using modelling languages such as UML or domain-specific Languages (DSLs) as source input, which contributes to saving time, effort and money.

This automatic generated code is then used for all sorts of things depending on the intended Purpose of the source modelling language and the target infrastructure. The task of creating This code is handled by a Generator which consists of a transformation engine (model source to code) and a runtime environment.

It can in some cases even add a high amount of functionality to the target that was not “modeled” in the source. Over time, software engineers have noticed recurring problems that can be analyzed so that we are able to conclude a general design pattern that helps along in the long run. The design pattern can accelerate the development process and provide proven development paradigms, which helps save time without having to reinvent patterns every time a problem arises whether it contributes to the reusability in the generator, or adaptability in the target code after it’s been generated.

Contents

1 Introduction

Generative software development is a principle that shares its dependence on design patterns just like many other principles in the software development paradigm. Generating any sort of code requires the use of a generator. A good generator, whether developed inhouse or one developed by a third party aims for reusability and adaptability. MontiCore [HoKR21] is a language workbench that generates tools by reusing existing components. As we will later see, MontiCore uses many design patterns for its implementation, and adapts some to create its own. This approach is fueled by the need of tools that are extendable and made from reusable components.

In this paper we present multiple design patterns that are implemented and adopted in a generative software example, namely MontiCore. We discuss how each of these patterns bring advantages that allow for reusability of already-existed code and adaptability of the code by a user of the generator.

We will start by defining key concepts that are needed for the structure of the paper. In chapter 2 we will discuss the generation gap pattern and its adoption in Monticore, namely the TOP mechanism and end the chapter with a comparison of the patterns respectively. Chapter 4 will introduce the Template hook pattern and how it’s used for both reusability and adaptability in MontiCore. Furthermore, in chapter 5, the decorator pattern will be introduced and similarly its usage in MontiCoreand also UML/P [Rum16] would be also presented. Finally, we will finish the paper with a conclusion that summarizes the patterns and their provided advantages in MontiCore.

2 Preliminaries

2.1 What is a Pattern

Expert designers know better than solving every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again, explained Gumma et al in their introduction [GHJV95]. Patterns in general are repeatable, identifiable sequences of elements or events. One cannot help but notice the use of patterns is in a wide area of cases and fields, whether we are planning to create new software systems, or design new buildings. For example, this is C. Alexander's description of patterns, but his book is in the context of buildings and towns: "*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*" [AIS+77]. This description is also true when describing patterns in the context of software development. Each of these design patterns have four main elements, the patterns identifying name, what problem is it solving, the solution that the pattern provides and lastly the consequences of implementing the pattern. Design patterns are crucial in the interest of well-written Code as they can be used in their basic implementation or they could be a foundation to base one's own patterns on, as they are adopted and adjusted to achieve required results. and are therefore of high importance when creating robust, maintainable code and saving time and effort.

2.2 What is generative software development

One of the biggest challenges in software development today is that software performance is increasingly at odds with programmer productivity. processor clock speeds are no longer doubling every as fast as before instead, computer hardware is rapidly becoming more and more parallel. Programming languages and methodologies focus increasingly on generality and abstraction, enabling programmers to build large systems from simple but versatile parts. This makes it difficult for compilers to translate high-level programs to efficient code, because they do not have the capability to really translate domain-specific operations. As an alternative to counting on a compiler that's smart enough to optimize a program, programmers can write a program generator that translates these these domain-specific operations into the code that we need. This concept of code generation is a good description of generative programming. P. Cointe describes it as a subdomain of metaprogramming and that they are an attempt to manufacture software components in an automated way by developing programs that synthesize other programs. [CO05]. Generative programing's goal is to improve the productivity of programmers. Two essential elements are generally needed to achieve the generation, namely a source Model and generator engine.

2.2.1 Model/DSL

A model is the library or framework that the DSL populates. It's a representation of the same subject that the DSL describes [MaFo10, page159-165]. Domain specific languages (DSLs) are programming languages that are designed for a particular domain. DSLs can be used to write code in any language, but they tend to have some special features that make them more convenient and easier to use than general purpose languages. DSLs are usually simple since they often have syntax that are easier to understand and closer to natural language than other programming languages. They also provide some built-in functionality so a developer doesn't need to create their own libraries or frameworks [MaFo10, page3-42]. DSLs can be used for a variety of goals:

- define commands to be executed
- describe documents or some of their specific aspects
- define rules or processes

DSLs generally have two kinds, internal and external. Internal DSLs are domain-specific languages that are defined using syntax that are different to that of the main programming language it's working with. The syntax may be simpler or in some cases more complex depending on the intended use. It will usually be parsed by the host application using text parsing techniques. On the other hand, internal DSLs gives developers a particular way of using general purpose languages (such as java or C). Developers would introduce a script in an internal DSL and it would be valid code in the general purpose language the DSL is working with, but it only uses limited features of the general purpose language in a particular style to handle small specified tasks.

2.2.2 Template engine

Template engine is a software component that facilitates the creation of textual output from a source model [MaFo10, page 539-547]. It's a software designed to combine templates with a data model to produce result files. A template engine is ordinarily included as a part of a web template system or application framework, and may be used also as a preprocessor or filter. The Idea behind a template engine is basically to generate output using a template which is the source text of the output text. Templates include callouts, that represent the dynamic parts of the template. These callouts are filled by the parts that vary upon generation and they reference the data model. The model represents the source of dynamic parts of the template and it serves as a context for the generation. A generator therefore acts as the component or tool that brings the template and model together resulting in dynamic generated output.

A good example for a template engine is the Apache FreeMarker template engine [Fre22]. FreeMarker is a freely available template engine which can easily be customized using the template files and changing data such as models. FreeMarker stores its templates as files

written in freeMarker template language, with extension "ftl". It contains parts of the general purpose language expressions that are Just like in classing These templates describe the general structure of the output to be generated in combination with an expression (model) and a control language that are evaluated when the templates are being processed. Figure 1 shows a representation of the FreeMarker functionality.

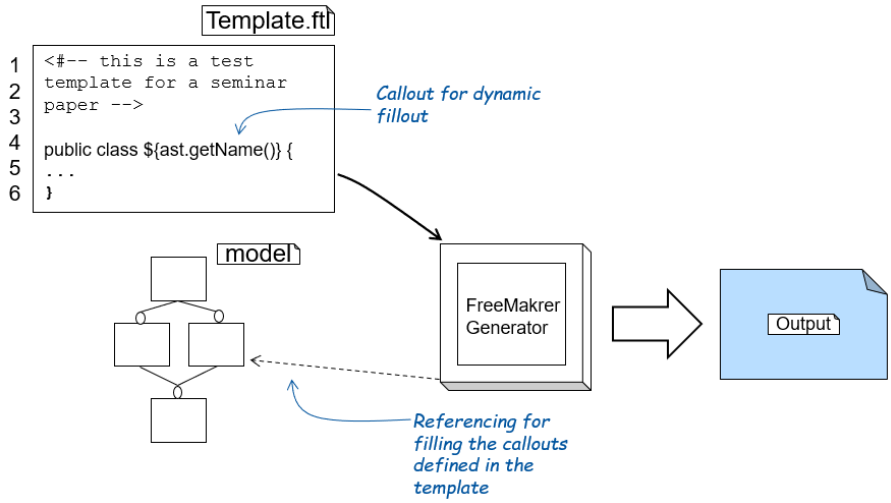


Figure 1: example representation of the FreeMarker template generator

This feature allows to write pieces of code in the general purpose language (such as java) with callouts in it, which will be filled upon generation by referencing a model. Figure 1 examples some of these expressions, namely ${} , and and comments like <#-- … --#>. Other forms of these callouts are control directives, such as <#if …>, and variable assignements with <#assign ...>. While FreeMarker was originally created for generating HTML pages in MVC web application frameworks, it isn't bound to servlets or HTML or anything web-related. It's used in non-web application environments as well, such as MotiCore [HoKR21, pp237-245]

2.3 What is MontiCore

MontiCore [HoKR21] is a language workbench for the efficient development of DSLs, developed by the Institute for Software Engineering of the RWTH Aachen university. M. Fowler defines a language workbench as a specialized software for defining and building DSLs. A language workbench is used not just to determine the structure of a DSL but also is a custom environment for developers to create DSL scripts [MaFo10]. MontiCore's goal is to help a developer create their own language tools. Some of these tools are generators themselves, which categorizes MotiCore as a meta-tool. It provides an array of features ranging from the modular definition of languages and language components in addition to

composing language tools, all the way to model analysis, transformation and the definition of tools for them

Some of MontiCore advantages are the reusability of predefined language components, conservative extension and composition mechanisms, and an optimal integration of handwritten code into the generated tools. Its grammar languages are comfortable to use. MontiCore provides sophisticated techniques to generate transformation languages and their transformation engines based on DSLs.

2.4 What is reusability

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. After talking about these concepts above, we somewhat get a sense of what this section is about. Reusability is one of the main motives that fuels the interest and research in generative software development. Several decades of research in software engineering left few alternatives but software reuse as the realistic approach to bring the gains in productivity and quality that the software industry needs, expressed H., F. and A. Mili in their opening introduction of [Mil95]. Reusability simply put, is the ability to reintegrate or reuse already defined software components for multiple purposes in order to save effort and time in implementation. Design patterns are a main ingredient in creating a system that takes advantage from already defined software. For example, decorator and template hook pattern as we will further see in more detail. Decorator will allow us to reuse decorator classes depending on the desired final design. Template hook pattern will give us ability to reintegrate and preexisting textual files into other to achieve different generated results.

2.5 What is adaptability

In generative programing, a lot of the times generated code needs to be extended with further functionality. Sometimes methods need to be added, attributes need to be supplied with. A good generator generates code with extension in mind. A user of that generator must be able to add the desired functionality without direct knowledge of the generator’s backend. This is what we mean by adaptability, it’s the ability to add and change functionality without the need to make changes in the backend. We will see this in the implementation of the template hook pattern for example as it allows for adapting the generated code simply by using hook methods. Similarly, TOP mechanism which is a special variant of the Generation gap pattern, also allows for adapting the generated code by finding a convenient way of integrating handwritten changes without having to change the related infrastructure of the modified classes.

3 GAP/TOP Pattern

Designing a generator that generates software from textual models such as DSLs or UML is a complex task that takes a lot of planning in the design stages. MontiCore's design was based on many design patterns, a main example for that would be template-hook which we will discuss later in this paper. There were other patterns however, that MontiCore has substantially adapted. This paved the way to create new patterns that are improved in the specific functionality that MontiCore needed. In this chapter we will discuss a design pattern that MontiCore based one of its main methods of adjusting generated code on, namely, the generation GAP pattern. We will discuss the intent behind using the pattern as well as its advantages and drawbacks. Later we'll introduce MontiCore's approach to integrate and improve the pattern which gave light to the TOP mechanism. We will end the chapter with a conclusion which includes comparing the two patterns.

3.1 GAP Pattern

Code generation from a model doesn't always results in the most optimal realization and often needs to be optimized to reach the desired results. Martin Fowler makes the point, when generating code from a model, textual or otherwise, we basically make that model the authoritative source for the generation [MaFo10, pages 571-579]. However, the prevailing conjecture is that deriving a non-trivial, complete implementation from models alone is not feasible, explained David Wile in [DaWi04]. If we go in and edit the generated code by hand, these changes would be lost upon regeneration. This also causes extra work on generation, which is not only bad in its own right, but also introduces a reluctant to change the DSL and generate again when necessary, undermining the whole point of a DSL. As a result, any generated code should never be touched by hand. This is where code generators express the need for integrating handwritten code that extends the generated artifacts to reach optimal results. Since we want the generated code to never be touched by hand, it makes perfect sense to keep them apart from handwritten code. Fowler's preference was to have files clearly separate into all-generated and all-handwritten. This is MontiCore's approach as well since it offers the crucial benefit of ensuring that the generator doesn't override handwritten code. Generation GAP is a software design pattern that was first documented by John Vlissides, motivated by the need to separate generated code from handwritten code [VlJo98]. The pattern has several variations when it comes to implementation, but all have the same goal which is a structure that separates generated code from handwritten code. These two structures are linked by inheritance. Vlissides's basic form of the pattern involves generating a superclass, which he refers to as the core class, and hand-coding a subclass for overriding any aspect of the generated code that needs to be changed. Consider the following code generation example adopted from [GHK+15]. Assume, we want to generate java classes from a source model while implementing the generation gap pattern. Assuming a class diagram, which contains the class Hospital is the source input for our generator. An optimal implementation of the methods of the Hospital class would have to be hand-coded and

integrated with the generated classes, since class diagrams don't model accurate class behavior. Generation Gap's approach would assume a generated superclass HospitalDefaultImpl for the class Hospital, which defines the general abstract functionality and a handwritten subclass HospitalCustomImpl which extends the superclass and contains any manual customization for specific methods in Hospital that need to behave different from the default implementation defined in HospitalDefaultImpl.

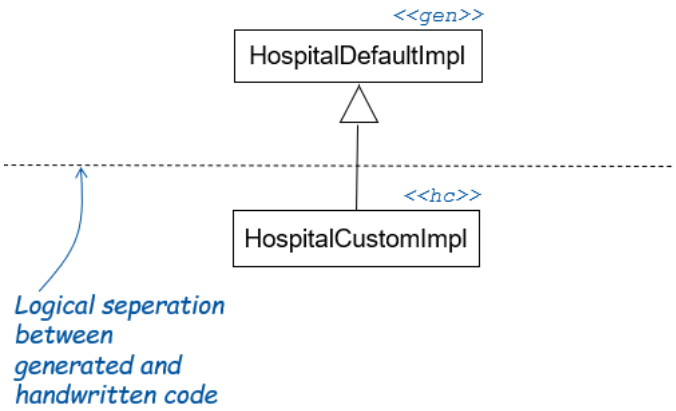


Figure 2: Generation Gap pattern for the class Hospital example

<<gen>> refers to generated class, <<hc>> refers to handwritten code

The Pattern ensures that any handwritten implementation added in HospitalCustomImpl wouldn't be lost upon regeneration since the generated superclass HospitalDefaultImpl's relation to the HospitalCustomIpl class is defined with inheritance. Any customized behavior defined in the handwritten class would simply override that of the superclass which results in verry efficient flexibility in the customization of the methods defined in the generated superclass. Now let's suppose the user of this generator wants to add methods to the Hospital class. Fowler makes the point, that when we refer to these classes from the outside, we always refer to the handwritten class that contains the concrete implementation. In this case that would be HospitalCustomImpl. This is where we notice one of the limitations the pattern has, as subclasses do not have the ability to add or remove methods of the generated parent class. Another issue arises, when we use the pattern for the generation of a system, in which handwritten extensions are rarely needed. In [GHK+15], it's explicitly mentioned that generation gap mechanism requires developers to create the handwritten class no matter whether handwritten code is inserted into it or not. In our mentioned scenario it would result in generating unneeded number of classes thus leading to a bloated project. MontiCore addressed these two issues in its implementation of the generation gap pattern which gave light to the TOP mechanism.

3.2 TOP Pattern

One of the disadvantages of generation gap in MontiCore, is the large overhead it causes when the generated artifacts are extended. As mentioned, Generation gap's method of code separation is achieved by subclassing. This approach is robust, but has the disadvantage that it is not possible to add new methods to the signature of the generated class directly, but to the subclass only. So either the subclass needs to be explicitly known in the rest of the system, or no additional functionality is available. This is why Drux, Jansen, and Rumpe argued in [DJR22] that an issue with generation gap is the necessity to integrate the handwritten subclasses at the using or instantiating locations of the generated code. Please keep in mind, that MontiCore is a framework which many of its handwritten subclasses in which theses extension reside, have corresponding builder classes. Builders are generated in form of the static delegator pattern. The static delegator is a design pattern that combines the advantages of publicly accessible static methods with the possibility to redefine them [GHJV95, page 221-223]. This is where the large overhead in noticeable. Even small changes in these subclasses would usually result in customizing the Builders as well.
TOP pattern is a pattern that was developed in MontiCore and it allows the automatic seamless integration of needed extensions in the generated classes. It's a less labor-intensive approach for developers as well, and it allows for extending the implementation, such as overriding methods, changing their signatures, or introducing completely new functionality. When extensions of a generated class are needed, the developer writes the desired class by hand and it would have the same name and package as the generated super class but it would be located in a dedicated path, in which the TOP mechanism is sensitive to handwritten classes. This handwritten class would be a subclass of the generated class. It alters the generation process such that an alternative class with the suffix TOP is generated instead which gives the pattern its name. However, if developers don't want to adjust the generated code, then a handwritten class doesn't have to be added. Instead, the generator checks at generation-time whether it exists. If it does exist, the generated class will extend the newly handwritten class. This addresses another issue the generation gap has, which is the necessity of having a handwritten class at all times. However, implementing TOP requires the generator to be executed again after adding a handwritten class to reflect this change in the generated code. Let's take the same Hospital example from earlier and apply the TOP-mechanism this time.

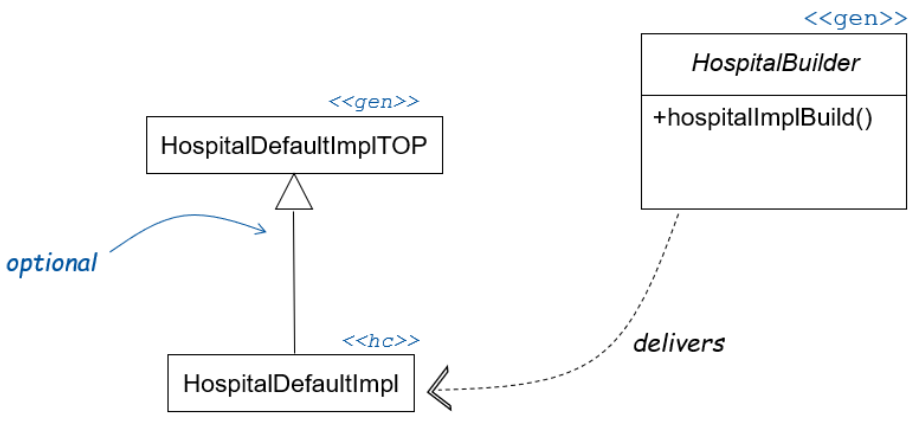


Figure 3: Top mechanism implementation for the hospital example.

In this scenario, we added a handwritten class HospitalDefaultImpl that contains changed and additional functionality to the source path of the generator. The generation process is now sensitive to the existence or removal of the classes in this path. So when we run the generator again, it notices the exitance of the handwritten class, and it will consequently generate the class containing the suffix TOP, namely HospitalDefaultImplTOP which contains identically generated code just with a different name. The handwritten class contains the implementation that's generated by default plus the new functionality added by hand earlier. However, the remaining infrastructure now points to the custom implementation (the handwritten class) since it has the old name of the generated class. This is what is meant by seamless integration of the handwritten code. HospitalDefaultImplTOP is actually now abstract. We referred to the inheritance shown in Figure 3 as "optional" because it could be evaded but it is recommended by the developers of MontiCore. In this scenario, HospitalDefaultImplTOP serves as a super class and it extends HospitalDefaultImpl. This is MontiCore's usual implementation and the reason for that has to do with the standard functionality that's additionally provided when generating, such as the HospitalBuilder class shown. Extending the generated artifacts makes the change effort minimal and applies only to the modified content. So in this case, HospitalBuilder returns instances of the handwritten class, as HospitalDefaultImpl extrends the generated artifact HospitalDefaultImplTOP, it directly inherits all required features and doesn't need any additional changes.

3.3 Comparison and Conclusion

Generation gap is a software design pattern that serves the main purpose of adaption of generated code without changing the generated files. Its basic functionality involves the generation of a super class in which the default implementation is introduced, and a handwritten subclass for the manual overriding of the default methods. This is a robust method that guarantees that that any handwritten changes to the inherited methods won't be lost when the generation process is ran again. This serves the adaptation concept introduced earlier in this paper. Besides that, the existence of handwritten code does not effect the generation process in any way since the generator would always requires the existence of a hand coded subclass. So in the case of any adaptation required that doesn't have the necessity of adding more methods or changing the inherited method's signatures, generation gap would be of great value. However, we discussed three main disadvantages the pattern has namely, the necessity for developers to have a hand coded subclass, the inability to extend the methods of the generated super class, and the large overhead caused by adaption in environments like MontiCore which generates additional artifact needed such as builders.

Generation TOP pattern was developed in MontiCore and is a special variant of the generation gap pattern. It offers a convenient solution for integrating handwritten artifacts into the generated infrastructure. When implemented, the generator would react to the existence of a handwritten class with the same name and would generate an abstract class with the a different name (with suffix TOP) allowing the reference on the handwritten class which results in the in automatic adaptation and integration of these manual changes

accordingly. This gives the ability to extend the generated class with more functionality, and not reduce the relation between the generated and handwritten class only to inheritance, although it's recommended in [DJR22]. Additionally, the TOP mechanism cancels the necessity of having an empty handwritten subclass since the generator would just generate the abstract class in case of unavailability of handwritten classes. In the case of constructor consistency between generated and handwritten classes, i.e. if the constructor for the hand coded class remains the same, then the generated builder can be reused directly. If the constructor of a hand coded class has, however, changed, the same TOP mechanism can be applied to the builder. A disadvantage of the TOP mechanism in comparison to generation gap would be the fact that independence of handwritten code at run-time is unfulfilled since the generator would always have to check whether a handwritten interface or a handwritten implementation class was introduced, as this influences the structure of the generated code.

4 Template Hook Pattern

Template hook pattern is a design pattern that belongs to the behavioral design patterns identified in [GHJV95, page325-331]. The intent behind developing such a pattern was the need to make the behavior of an algorithm that's running more flexible in its design and implementation while avoiding duplicate code and unneeded effort in reimplementation. The main idea of how template hook pattern's implementation works is very basic, it defines the general behavior of an algorithm and the steps to execute it in a super class which can provide default implementation that might be overwritten by the subclasses that needs the algorithm to act in a more specialized matter. Most of the times, subclasses call methods from super class but in template pattern, superclass template method calls methods from subclasses, this is known as Hollywood Principle - "don't call us, we'll call you." [MiMa96].

Consider the following example: Let's suppose we want to provide an algorithm to write a seminar paper like the one I'm presenting. The steps needed to write a seminar paper are, find a topic, research the idea and lastly start with the writing and drawing. It's important to keep in mind that we can't change the order of execution without introducing hook methods (which we will discuss later) but that's convenient in our example, because if we think about it, we can't start with researching if we don't know what it is exactly, we're researching about so we need a topic before building a solid research foundation and then we can start with composing of the paper. In this case we can create a template method in which a general algorithm, for example composePaper() is specified and in it exists the steps or in other words methods needed to implement this algorithm like for example determineTopic(), cite(), writePaper() and drawPaper(). Naturally the subclasses that use these methods have 2 options, either use the default implementation or overwrite what needs to be overwritten. Now for the sake of argument, let's suppose that the Matse-group already prepares a pool of topics of which the students have to choose from. This will make the method of finding a topic the same for all types of seminar papers, whether it's a concept, or it's in English, German, made with word, latex and so on. We can provide base implementation for this by restricting subclasses from overriding this method (step), and in this case we do that simply by making that method *final*.

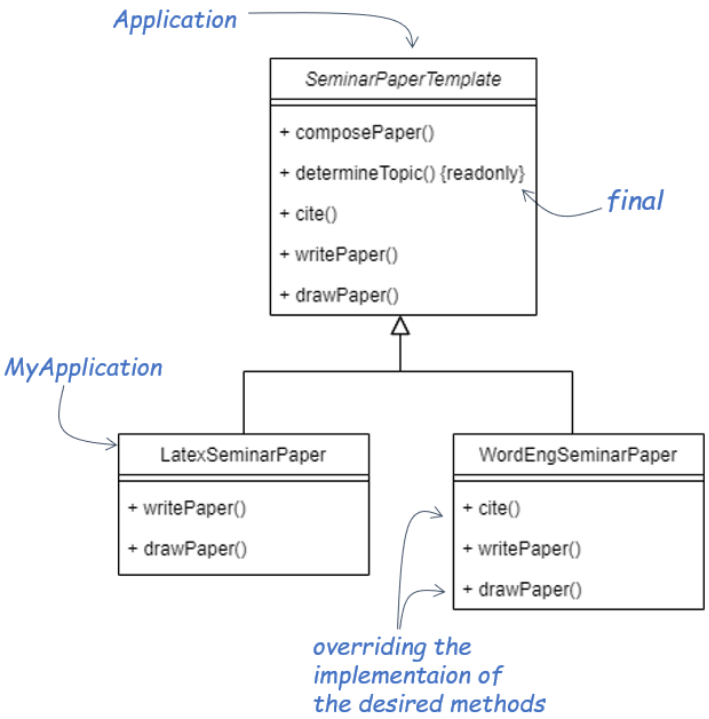


Figure 4: class diagram that shows the basic functionality of template hook pattern.

Up next, we will discuss an example usage of the template hook pattern in a generative software development example, namely MontiCore. We will get to know how Template hook is a pattern that MontiCore utilizes for both reusability and Adaptability. Finally, we will connect all the points together with a conclusion.

4.1 Template-hook for reusability

The pattern consists of two methods: the template method and the hook method [HoKR21]. The template contains the general logic that defines the algorithm in an abstract way. It calls the specific functionality using the hook method. The hook method is therefore the principle of the template's calling of the more specific functionalities that do not exist in it. We use hooks to override partial implementation which allows the customization of the algorithm without the need to redevelop an entire algorithm. One of the methods MontiCore uses to generate code, is the TOP mechanism that we introduced in chapter 3.2 . We explained how TOP can utilize subclassing to adapt generated code. If the generated TOP-class has a Template method, then MontiCore applies this pattern in subclassing with the TOP mechanism that to customize the generate code.

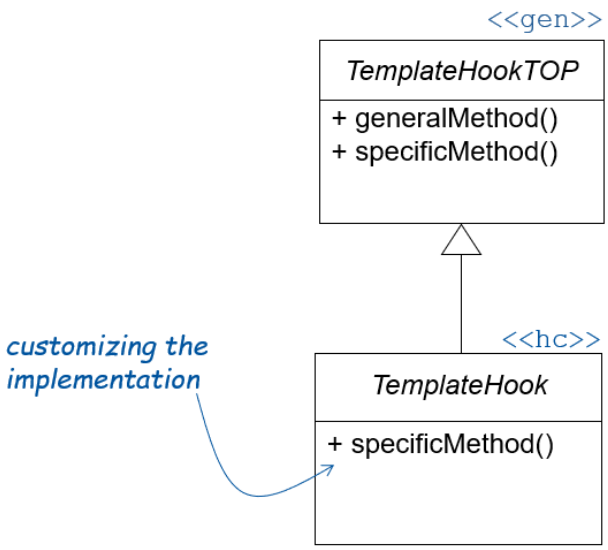


Figure 5: Template hook pattern being used for integrating handwritten and generated code adopted from [DJR22]

Figure 4 shows a generated TemplateHookTOP that's being extended by a handwritten CostumHook that's used for customizing a specific method using the template hook pattern. The generalMethod() method calls specificMethod() -which acts as the hook method- and the handwritten CustomHook overrides the implantation using subclassing. This conjunction of core functionality and delegation of some basic actions to hook methods is a key mechanism in frameworks, where the template method belongs to the framework and the hook method is meant to be defined by the individual application via subclassing respective framework classes

Another method MontiCore uses to generate java artifacts is the via FreeMarker template engine. We discussed in Chapter 2 that freeMarker uses templates to store the intended general structure of the code, that is to be generated. Additionally the define callouts that are dynamically filled by referencing the desired data model (context). In these freeMarker templates, we also can specify what's known as hook points. A summary of hook point's definition mentioned in [HoKR21, page245-281] would be, a place in the freeMarker template that's meant for customizing the generated artifacts. The hook point consists of a name, which identifies the place in the template, where to hook in, and a value, that is bound to the hook point. A hook point could be defined using the include command, then the template itself has static knowledge of the included template and no additional binding (e.g. in the controller) is required. This allows hook points to be defined implicitly in these freeMarker templates. Hook points can also be defined explicitly by using the "definHookPoint" command, but only if *bound* to a value or they would default to an empty String. This is where we make use of the template method pattern for reusability as well. Because of the pattern's compatibility with the hook points that exist in the freeMarker templates, we can reuse templates that we already bound in other freeMarker templates for different purposes. This saves the effort of redeveloping templates and by that displaying another reusability feature of the template hook pattern.

4.2 Template-hook for adaptability

Drux, Jansen, and Rumpe [DJR22] argue that, generators created with MontiCore are designed explicitly for providing various hook methods. The reasoning behind this architecture is. This is due to fact that in some cases generated code is intended to be extended by handwritten code or at least an option to extend generation using handwritten code should be provided. In this case, providing such hooks leads to flexibly extensible generated code. So it makes sense to design a generated piece of code similar to a framework providing various hook methods. In the same paper, a none-standard way of integrating the template hook method is discussed, namely by relocating the hook methods to hook classes that are separate from the template class. This is similar behavior the delegator pattern [GHJV95, Intro] where two objects are involved in handling a request: a receiving object -in this case the Template class- delegates operations to its delegate, namely the hook class. The main advantage of delegation is that it makes it easy to compose behaviors at run-time and to change the way they're composed. Therefore, the hook methods can be interchangeable even during run time, due to the total separation between the Template class and the hook class. The Template would always implement the same behavior and the customization via hook calls would be delegated to the sperate hook classes. The main disadvantage Delegation has, one it shares with other techniques that make software more flexible through object composition: Overhead. Highly parameterized, dynamic software is harder to understand than more static software. Delegation is a good design choice only when it simplifies more than it complicates.

Another integration of the pattern for adaptability is noticeable in MontiCore's usage of the FreeMarker template engine as well. In MontiCore, we extend the FreeMarker engine with a controller that integrates the template hook pattern throuth its definition of explicit hook points. Figure 6 shall help with explaining this mechanism:

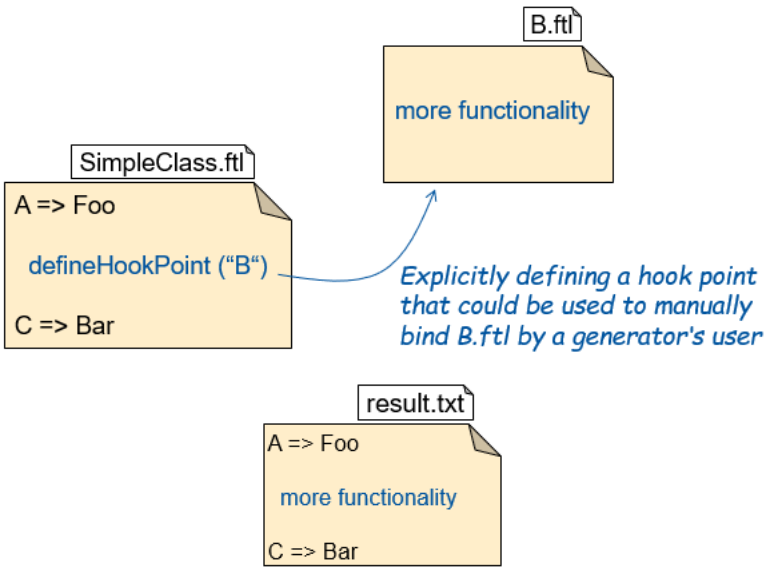


Figure 6: explicit hook points used to flexibly add more functionality

Figure 5 represents a simple FreeMarker template. We define in the body of the template an explicit hook point with the command “defineHookPoint(“B”)”. This hook point in the example is still empty and is not bound yet by the user of the generator. It could, however, be bound manually to one or multiple templates that add more functionality. In the example B.ftl is just there to show the concept of manually adding more functionality by the generator’s user through explicit hook points. If bound, the generator would translate the content of the template B.ftl and would add the result to SimpleClass.ftl in the specified order. B.ftl could result in adding more attributes, methods or any adjustments the user of the generator sees fitting. The user could also bind multiple templates using the same mechanism. This separation of generated abstract classes and concrete handwritten implementation is what makes template hook pattern so well integrated in MontiCore, since it facilitates the separation of concerns [HuLo95], which is a core concept in MontiCore’s method of code generation, also this allows the developers to regenerate without losing the handwritten extensions.

To tie things together again, template-hook pattern is a powerful design model to take advantage of when common behavior among subclasses with noticeable differences should be factored and localized in a superclass to avoid code duplication. This is a good example of "refactoring to generalize" as described by Opdyke and Johnson [OJ93]. We saw how the pattern’s implementation in a simple real word example such as the seminar paper scenario discussed at the beginning of the chapter is similar to MontiCore’s approach, however we also witnessed MontiCore’s more advanced implementation of the pattern in its generation mechanism. Two different variants of the pattern’s implementation were introduced, integrated template-hook class and. We saw TOP mechanism’s integration of the pattern for reusability by overriding methods by subclassing without redeveloping entire algorithms. Additionally, TOP mechanism’s method for adaptability by integration of another variant of template hook similar to that of the delegator pattern were template class delegates to a separate hook class which made the hook’s behavior interchangeable even during runtime (although with more overhead). We discussed the pattern’s integration in the FreeMarker engine, implicitly which allows reusing templates to generate different results and explicitly which allows both the generated infrastructure and the generator to be extremely adaptive to extensions.Both template and hook methods work hand in hand to facilitate the separation of concerns whether it’s from the developer side in saving effort and time by shifting the functionality elsewhere, separating it from the abstract super class or from a user’s side by keeping the used templates flexibly interchangeable during run time. It’s one the patterns that serve both Reusability and adaptability.

5 Decorator Pattern

In this chapter, we will talk about a design pattern that belongs to the structural patterns, namely, the Decorator pattern [GHJV95, p175-185]. The intent behind the pattern is to add more functionality to an object at run-time. Usually, we use inheritance or composition to extend the behavior of an object, but this is done at compile time and its applicable to all the instances of the class. We can't add any new functionality of remove any existing behavior at runtime. This is when Decorator pattern comes into picture. Let's take an example. Suppose we want to implement different kinds of house decorations. If we implement this using inheritance, we would create an interface House to define the decorate() method and then we can have a basic HouseDecoration class that implements it. We can then extend HouseDecoration with more classes that define deferent types of decorations that we might need. For example ChristmasDecoration and ChildrenDecoration. If we want to get a decoration at runtime that has both the features of Christmas and children, then the implementation gets complex. If furthermore, we want to specify which decoration should be added first, it gets even more complex. This inheritance structure is only going to get more and more complex and difficult to manage the more decorations and conditions we add. To solve this kind of programming situation, we apply decorator pattern. The following Figure 7 shows the same scenario but with a decorator implementation.

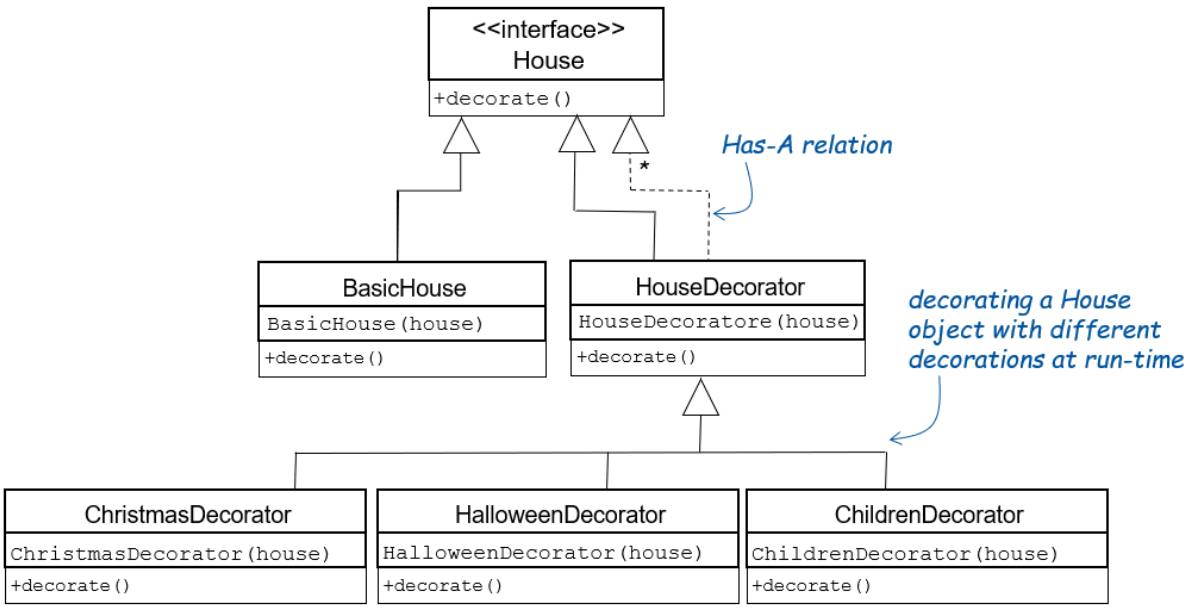


Figure 7: implementation of the decorator pattern in a house example

First we create an interface defining the method decorate() that will be passed down. In our case House will be the component interface. Then we define the basic implementation BasicHouse of the component interface. The Decorator class HouseDecorator implements the House and it has a "HAS-A" relationship with the component interface. The variable that's defined in HouseDecorator should be accessible to the child decorator classes, so in

this case we make this variable protected. Then we extend the HouseDecorator class which is the base decorator functionality with as many concrete decoration-classes as needed (ChristmasDecorator, HalloweenDecorator, ChildrenDecorator) and we modify the component behavior accordingly. This way, the client program can create different kinds of Objects at runtime and they can specify the order of execution too. With inheritance an object's functionality is extended by adding a subclass at compile time. Decorator pattern's implementation wraps the component object (house) dynamically with added functionality or responsibility at run-time without the object's knowledge of the decorator's existence. This is why it's referred to as "*Wrapper*" in [GHJV95, p175].

In MontiCore [HoKR21, page245-274] we apply the Decorator pattern in decorating templates before generation to add more functionality without dependence on inheritance and without the template knowing it's being decorated. We talked in chapter 4 about the controller that we extend the generator with, in order to manage the Freemarker templates. There we described the concept of a hook point, and we mentioned different types of that concept. In this situation we also depend on the usage of hook points defined in templates (both implicitly or explicitly), but in this case we use a different binding mechanism to decorate the templates before generation. The class that provides the binding method introduced in chapter 4, also provides decorating methods, with which functionality is added in a specific order. Methods like setBeforeTemplate() or setAfterTemplate() take other hook points or templates -a large number of are designed for decoration- as arguments and they would be used to decorate the hook points specified. Once these methods are implemented, the controller would take advantage of already written decorator-templates to add more functionality in a specific order needed. Let's take an example Adopted from [HoKR21]. Suppose we want to generate java artifacts using a UML state diagram as a source-model. We're using a state diagram since it provides the language definitions for advanced tooling such as model-transformation. The state diagram would be realized by a set of templates each realizing different elements of the diagram, for example, abstract state, concrete classes, attributes etc. In most cases, MontiCore uses these templates as a part of a fixed library that's intended for reusability and are not to be changed each time a generation is needed. Figure 8 shows a standard generation example for attributes using the standard templates defined in the controller manager:

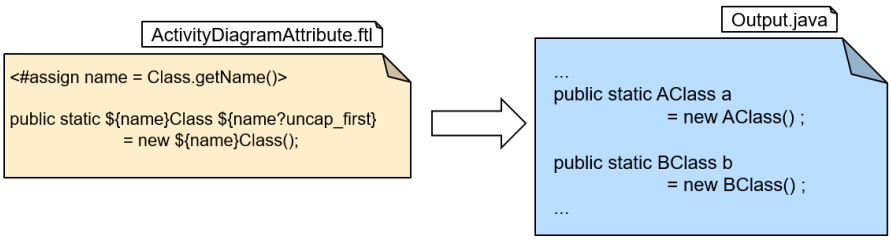


Figure 8: generation of attributes using a standard template without decoration

Now suppose we want to add access functionality to the attributes that are generated by providing getter methods for each generated attribute. This would be a great opportunity to decorate the ActivityDiagramAttribute.ftl template with a decorator template that we design specifically for access functionality. In this case, we would need to define a hook point in the

template controller to attach the needed template. Finally we bind the decorator template using the setAfterTemplate(ActivityDiagramAttribute.ftl, AttributeGetter.ftl) command in this case, that generates getter methods after the attributes are declared respectively. Figure 9 shows the resulted output after we decorated the template with more functionality:

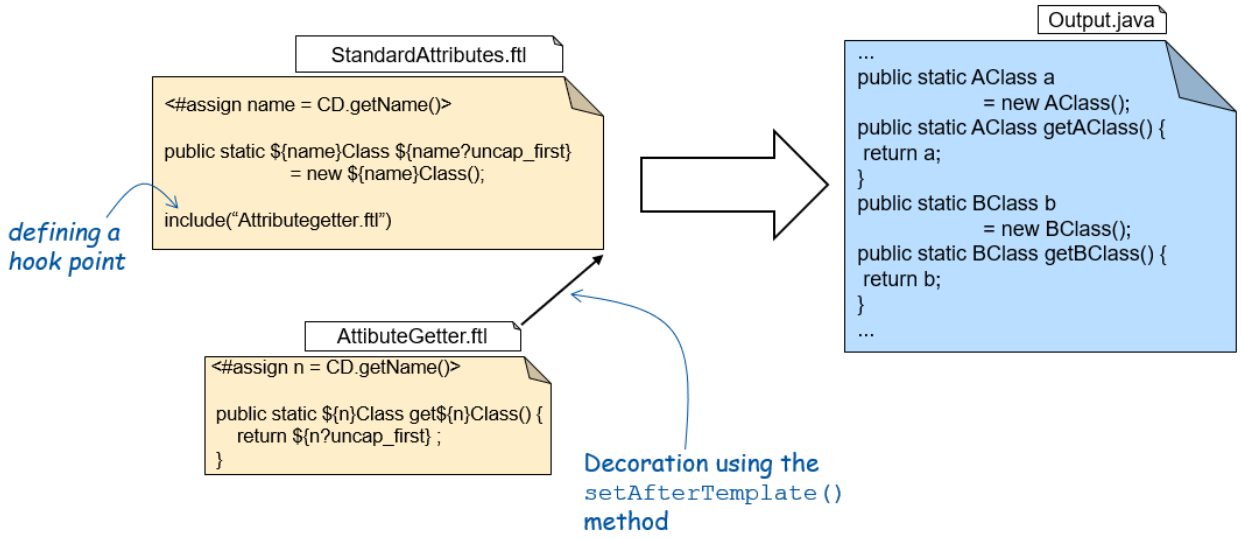


Figure 9: binding of attributes using a standard template with decoration

We could reach a similar result if we use the include("AttributeGetter.ftl") method since it also decorates the template with another template but there is a main difference between that method and the setBeforeTemplate or setAfterTemplate. Method Include() enforces the hosting template's static knowledge of the template that's being included. For example, if we type the command include("B.ftl") in template A.ftl, then A.ftl would have static knowledge of B.ftl . With the setBefore or setAfter methods then the knowledge direction would be inverted and in the same example if we specify to A.ftl to be decorated with B.ftl after via the command setAfterTemplae("A.ftl", "B.ftl") then A.ftl wouldn't have any knowledge of its decoration with B.ftl .

Another application of the Decorator pattern is in metamodel extendibility mechanism [TrGa10]. A metamodel defines the structure (abstract syntax) of a modeling language. With nodes and edges. A metamodel defines syntactical rules to which the instance model must conform [VFV06a]. A good example for that is UML/P. B. Rumpe, defines UML/P as a specifiable language that consists of several types of diagrams and texts that could be used in an integrated form [Rum16]. It's a language profile that consists of 6 sub notations such as class, sequence and object diagrams among others. It's used as a notation for a number of activities such as use case modeling, target performance analysis, as well as architectural and detailed design at different levels of granularity. Model extendibility in UML/P is a proactive approach in which the metamodel engineers recognize that the attachment of additional information to model elements might be needed or desired. Therefore, UML/P is designed with extendibility in mind. Its design provides built-in support for attaching arbitrary key-value pairs of information to any element of the model. These pairs are referred to as

stereotypes and tags. Stereotypes and tags are used to extend and adjust the language vocabulary according to the respective needed requirements. We use the decorator pattern to decorate any element in the model using stereotypes and tags [Rum16]. Stereotypes generally have a multitude of (overlapping) application possibilities depending on the scenario and the stereotype used. For example, the stereotype "*refine*" in the UML standard is designed for describing a methodical relation between model elements. stereotypes of the form "*Wrapper*", allow the role of a class in a design pattern to be documented. In general Stereotypes can also demand additional properties from model elements as well as specialize existing properties, so they have the ability to describe syntactic properties of a model element. Another form of application is ability to describe application-specific requirements, a food example for that is the "*persistent*" stereotype, which can specify that objects of this class are persistently stored. much more. Tags can be attached to basically to each model element to allow the developer to specify properties in a more detailed form. This helps with solving the problem of overlapping functionality of different types of stereotypes since one or multiple tags could also be bound to a stereotype. Whenever more specification or different functionality is required in a model element, developers would use one of these already-defined stereotypes in combination with one or multiple tags if needed to decorate and manipulate the model's different elements as desired. This means the same stereotype could be used for multiple model elements which contributes to the reusability of preexisting code. [Rum16] [TrGa10].

In this chapter we discussed the functionality of the decorator pattern. We started with explaining the pattern's functionality by using a real-word example. We then followed it with two adaption implementations in generative programming, namely in MontiCore and UML/P. We explained how The pattern's implementation was demonstrated in the MontiCore generator by decorating the Freemarker templates by taking advantage of the hook point concept which would allow the templates to be decorated without their static knowledge of the decorating templates. This allowed for the reusability of other templates. Meanwhile in UML/P we argued how the decorator pattern was giving the advantage of reusability in metamodel extension using decoration of model elements, by reusing stereotypes and tags that would manipulate elements of the model.

6 Conclusion

Design patterns are powerful tools, with which developers can reach robust, optimal and clean implementation. Each of these patterns was found with the motive of solving a noticeable, recurring problem in a specific situation. They have been implemented in countless situation with all of their characteristics documented and proven over time to be accurate. In this paper we presented a number of design patterns used in generative software development. We used [GHJV95] as a citation for them throughout the paper since it's one of the main references for software design patterns. Our main example for the patterns implementation in a generative environment was the language workbench MontiCore [HoKR21] since it was designed on a solid foundation of well implemented design patterns as well as in-house design patterns. For each of patterns discussed, we showed an example usage in a traditional software design and then followed it with at least one example usage in MontiCore. We defined some of the main concepts needed for the comprehension of the paper and followed them with a discussion about four different patterns and focused on the reusability and/or adaptability they contributed to the system.

Sarting with generation gap. Generation Gap is a pattern for integrating generated and handwritten code, where generated and handwritten code would be guaranteed to stay separated and regeneration won't delete the handwritten extensions, however its disadvantage was the lack of customization of the generated classes and the overhead it could introduce in generative environment that provide other functionality as well as the necessity to always generate a handwritten class. This was addressed by TOP which is a MontiCore-developed pattern also used of integrating generated and handwritten code. TOP mechanism allows the generator to detect handwritten classes and would incorporates handwritten classes instead of generated classes, which solves the extension problem and the necessity for handwritten classes that Gap had. It also didn't depend solely on subclassing which solved the overhead problem. We saw how both of these patterns were achieving adaptability, but TOP was giving more adaptability since seamless integration of handwritten code was achieved with no additional integration effort was needed.

Then we discussed the Template hook pattern. Template hook was mainly used to split generation steps into multiple modular functions. We argued how it achieved both reusability and adaptability. Its integration in the FreeMarker templates by using hook points to reuse a template to generate different target results as well as facilitating extension by the definition of explicit hook points which would be used to bind other templates by an end user. We saw how the pattern also cooperated well with TOP in extension scenarios where we would implement a template and a hook method in the generated class, allowing the handwritten class to extend the functionality by overriding the hook method. Furthermore, we presented a reference to delegator pattern, since a variant of template hook can implement the original parts in sub-classes of the delegates.

Finally, we discussed the decorator pattern and argued that it served mainly the reusability concept. Decorator's intent is to add more functionality at run-time. In MontiCore we took advantage of the hook points defined in templates to decorate a template using a binding mechanism that lets the decorated template to be blind of the decoration. This allowed for reusing of decorator templates and achieved reusability. Another example of the pattern was in the UML/P language, where the decoration of the model was achieved by adding already-defined stereotypes and binging them with tags to add more detail or functionality to different model elements, facilitating the reuse of these software components (stereotypes and Tags).

With these discussions and points, we argued how each of the above-mentioned patterns was used by MontiCore to serve a certain role. We were mainly focused on reusability such in the case of template hook and decorator and adaptability in the case of Gap/TOP and also the different implementation template hook.

7 Bibliography

[AIS+ 77] Christopher Alexander,Sara Ishikawa, MurraySilverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. A Pattern Language. Oxford University Press, New York, 1977. Page 10

[Co05] Cointe, P. (2005). Towards Generative Programming. In: Banâtre, JP., Fradet, P., Giavitto, JL., Michel, O. (eds) Unconventional Programming Paradigms. UPP 2004. Lecture Notes in Computer Science, vol 3566. Springer, Berlin, Heidelberg. Pp. 315-325, doi.org:10.1007/11527800_24

[DaWi04] Wile, D. S. (2003). Lessons Learned from Real DSL Experiments. In Proceedings of the 36th Annual Hawaii International Conference on System Sciences, HICSS ’03, pages 265–290. IEEE Computer Society

[DJR22] Florian Drux, Nico Jansen, Bernhard Rumpe, “A Catalog of Design Patterns for Compositional Language Engineering”, Journal of Object Technology, Volume 21, no. 4 (October 2022), pp. 4:1-13, doi:10.5381/jot.2022.21.4.a4.

[Fre22] FreeMarker website. http://freemarker.org/, 2022. Retreaved on 11.12.2022

[GHJV95] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Design Patterns. massachusetts: Addison-Wesley Publishing Company

[GHK+15] T. Greifenberg, K. Hölldobler, C. Kolassa, M. Look, P. Mir Seyed Nazari, K. Müller, A. Navarro Perez, D. Plotnikov, D. Reiss, A. Roth, B. Rumpe, M. Schindler, A. Wortmann: A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development. Angers, Loire Valley, France, pp. 74-85, 2015.

[HoKR21] Hölldobler, K., Kautz, O., & Rumpe, B. (2021). MontiCore Language Workbench and Library Handbook: Edition 2021. Shaker Verlag pages

[HuLo95] Hürsch, W. L., & Lopes, C. V. (1995). Separation of Concerns. Kleppe, A. (2008). Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Pearson Education

[MaFo10] Martin Fowler. 2010. Domain Specific Languages (1st. ed.). Addison-Wesley Professional, pages 571-579

[Mil95] H. Mili, F. Mili, A. Mili, Reusing Software: Issues and Research Directions, IEEE Transactions on Software Engineering, Vol. 21, No. 6, June 1995

[MiMa96] Mattsson, M.. Object-oriented frameworks. *Licentiate thesis*, pp. 98, 1996.

[OJ93] William F.Opdyke and Ralph E.Johnson. Creating abstract superclasses by refactoring. In Proceedings of the 21st Annual Computer Science Conference (ACM CSC '93), pages 66-73,Indianapolis, IN, February 1993

[Pre95] W. Pree. Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995.

[Rot17] Alexander Roth. Adaptable Code Generation of Consistent and Customizable Data Centric Applications with MontiDex. Aachener InformatikBerichte, Software Engineering, Band 31. Shaker Verlag, December 2017.

[Rum16] Bernhard Rumpe. Modeling with UML: Language, Concepts, Methods. Springer International, pp. 231-257 July 2016.

[TrGa10] Kolovos, D.S., Rose, L.M., Drivalos Matragkas, N., Paige, R.F., Polack, F.A.C., Fernandes, K.J. (2010). Constructing and Navigating Non-invasive Model Decorations. In: Tratt, L., Gogolla, M. (eds) Theory and Practice of Model Transformations. ICMT 2010. Lecture Notes in Computer Science, vol 6142. Springer, Berlin, Heidelberg. doi.org/10.1007/978-3-642-13688-7_10, pp. 138–152

[VFV06a] Gergely Varro, Katalin Friedl, and Daniel Varro. Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. Electronic Notes in Theoretical Computer Science, 152:191–205, 2006.

[VlJo98] Vlissides, John (1998-06-22). *Pattern Hatching: Design Patterns Applied*. Addison-Wesley Professional. pp. 85–101. ISBN 978-0201432930.