A Catalog of Design Patterns for Compositional Language Engineering

Florian Drux[*], **Nico Jansen**[*], **and Bernhard Rumpe**[*]
[*]Software Engineering, RWTH Aachen University

ABSTRACT When composing a domain-specific language from several language components, it is also necessary to compose analysis and synthesis techniques, which are individually defined on these components in an efficient, ideally black-box form. An effective way of allowing such compositions is to use specific design patterns, which are partly reflected in the tooling code, partly reflected in the language, but also partly reflected in the language workbench (one meta-level higher), and the generated/synthesized product code (one meta-level downward). Based on the experiences gained in compositional language development using the language workbench MontiCore, we in detail discuss several of those design patterns, namely the Mill, the RealThis object composition, the Template/Hook, and the TOP- Generator Patterns, and the hidden complexity of an extended visitor infrastructure coping with the above patterns. The patterns are recorded and described in a reusable way, as usual, allowing readers to participate from the gained insights and possible solutions.

KEYWORDS Software Language Engineering, Language Composition, Design Patterns

1. Introduction

Model-driven development (MDD) (Selic 2003) gains more and more significance in modern engineering of various domains such as software development (Völter et al. 2013), avionics (Feiler & Gluch 2012), automotive (Blom et al. 2013), and robotics (Wigand et al. 2017). It promotes models as the primary artifacts in the development process (France & Rumpe 2007) that describe the system to be engineered. Here, automation is key to foster efficient and agile design and manufacturing. However, to realize this automation, models must be in machine-processable form, related to a structured syntactic format and a distinct meaning (Harel & Rumpe 2004) of the individual constituents. Thus, models must adhere to modeling languages specifying the concrete and abstract syntax, the semantics, a semantic domain, and a semantic mapping from this domain to the language elements (Cengarle et al. 2009). Generally, we distinguish between general-purpose languages (GPLs) used across disciplines (such as UML (Rumpe 2016) or SysML (Friedenthal et al. 2014)) and domain-specific languages (DSLs) tailored for a specific application domain.

JOT reference format:
Florian Drux, Nico Jansen, and Bernhard Rumpe. *A Catalog of Design Patterns for Compositional Language Engineering*. Journal of Object Technology. Vol. V, No. N, 2011. Licensed under Attribution - No Derivatives 4.0 International (CC BY-ND 4.0) http://dx.doi.org/10.5381/jot.2011.V.N.M

The discipline for efficiently designing, implementing, maintaining, and deploying such languages is known as software language engineering (SLE) (Kleppe 2008). There are various techniques for developing modeling languages. Here, two prominent approaches are repeatedly applied: Language development via context-free grammar (Klint et al. 2005), which defines concrete and abstract syntax in a single effort, and via metamodels (such as Ecore (Steinberg et al. 2008)) for determining the abstract syntax. The first approach often occurs for specifying textual languages. In contrast, the second approach can be applied to both textual and graphical languages; however, being more prominent for the latter.

Since the developed modeling languages are software too (Favre 2005), they also suffer from the same issues as programs in ordinary software engineering. This means they are subject to constant refinement, maintenance and evolution. Additionally, starting from scratch is usually time-consuming and tedious. Especially when it comes to implementing already known concepts, the reuse of code via libraries has proven to be highly beneficial in software development, increasing the development efficiency as well as the quality of the software. This notion is also gradually finding its way into the discipline of SLE (Bryant et al. 2015), where re-using (partial) languages has great potential, e.g., for implementing different UML profiles (Fuentes-Fernández & Vallecillo-Moreno 2004).

Generally, there are different kinds of reusability in modeling languages. The most rudimentary form occurs at a conceptual level, where languages support similar concepts (Medvidovic & Taylor 2000) or, in some cases, contain similar syntactic constructs. However, there is no actual reuse here, as these are entirely individual development processes. Desirable would be the reuse of the actual language definitions (i.e., grammars or meta-models), as well as the existing tooling such as well-formedness checks, analyses, or implementations for synthesis.

Thus, the inter-language reusability of constituents from different technological spaces (Kurtev et al. 2002) is essential for enabling language engineers to focus on more sophisticated details. In this context, the composition of distinct modeling languages (Hölldobler et al. 2018) and establishing reusable language libraries (Butting et al. 2020) are increasingly explored. While various composition mechanisms have been investigated across different technological spaces, a sustainable design of the underlying (generated) infrastructure is essential. To tackle this challenge in traditional software engineering, the community of object-oriented software development established various design patterns (Gamma et al. 1995). Applying these patterns assists in the efficient development of reusable and maintainable software systems.

To harness the benefits of such patterns to the discipline of SLE, in this paper, we elaborate on a catalog of design patterns specifically tailored to language composition. These rely on new practices as well as on modifications of existing patterns dedicated to the distinct SLE challenges. We present our realization within the MontiCore language workbench (Hölldobler et al. 2021) and report our observations on how these designs facilitate compositional language engineering with respect to enabling black-box reusability of the generated and handwritten tooling.

The remainder of this paper is structured as follows: Section 2 introduces preliminary work. Section 3 describes our catalog of design patterns for compositional language design. Section 4 discusses the advantages and disadvantages of applying the elaborated patterns, while Section 5 considers related work. Finally, Section 6 concludes our work.

2. Preliminaries

Our work is based on foundations and discoveries within the MontiCore language workbench and its unique features for language composition. This section introduces both.

2.1. MontiCore

MontiCore (Hölldobler et al. 2021) is a language workbench for the efficient conceptualization, design, implementation, and extensibility of modeling languages. These languages are created based on context-free grammars in an EBNF-like (Wirth 1996) syntax. Thus, the concrete and the abstract syntax of MontiCore languages are created in a single effort.

When creating a new language, MontiCore processes the input grammar and generates the foundational Java infrastructure. The generated artifacts include a parser, abstract syntax classes, a framework for well-formedness rules, symbol management support, a visitor infrastructure, and code generation framework. The parser processes textual models and transforms them into an internal data structure, the abstract syntax tree (AST), by instantiating the corresponding abstract syntax classes. The AST is the abstract representation of a model, cleansed of syntactic sugar. It is the primary data structure on which checks, transformations, and synthesis are performed.

To check the well-formedness of models, MontiCore provides a so-called context condition (CoCo) framework. It enables performing additional validation checks on the AST. These are primarily context-sensitive tests or those which are difficult or impossible to capture using the parser of a context-free grammar. A common example is validating whether a used variable has been declared before or whether it is set with a value of a correct data type. Accordingly, CoCos are subsequent checks that further restrict the set of valid models.

For each language, MontiCore comes with an infrastructure for efficient management of symbols, the symbol table. The general concept is adopted from compiler construction and provides functionalities for efficient symbol resolving. Each element with a unique referencable name is a symbol. The symbol table thus contains the essential elements of a model, which are usually also externally accessible (e.g., comparable to types, fields, and methods in object-oriented programming languages). It allows for efficient cross-referencing of elements and thus lifts the tree structure of the AST to a graph structure.

For the efficient and type-safe traversal of AST and symbol table, MontiCore supports a realization of the visitor pattern (Gamma et al. 1995). It offers the general capability to provide custom functionality for distinct aspects of the modeling language without anchoring them directly in the model. The visitor infrastructure is thus the foundation for implementing custom operations and analyses. Multiple standard functionalities of MontiCore are based on this infrastructure, such as the CoCo framework or the symbol table.

MontiCore provides a code generation framework by default. It processes the input AST and translates it into executable program artifacts. For this purpose, the FreeMarker template engine (FreeMarker 2022) is used. A template usually operates on an AST node and represents the structure of the target code to be synthesized. In addition, MontiCore builds a framework around the FreeMarker engine, which supports managing the generator templates, as well as configuring it from the outside via hook points.

Within the MontiCore language workbench, some research on design patterns for SLE has been conducted. These are discussed in this paper under the particular background of compositional language engineering.

2.2. Language Composition

In object-oriented software engineering, best practices consider the modularization of individual software components. These can be composed differently and thus be reused. This notion gets more and more applied to SLE in the large (Hölldobler et al. 2018) to cope with the ever-growing complexity when developing bigger and more sophisticated modeling languages (Vallecillo 2010). Therefore, many state-of-the-art language workbenches feature various techniques to enable the composition (Erdweg et al. 2013) of existing language components. A language component is a self-contained but possibly incomplete logical unit of a language definition that is explicitly designed for reuse and integration (Clark et al. 2015). An intuitive example is a library of reusable expressions, literals, statements, or types (Butting et al. 2020). Still, also full-fledged languages can feature integrability, such as Statecharts as behavior description language inside a component and connector architecture (Butting et al. 2016).

To support sophisticated SLE, MontiCore also offers different approaches to language composition (Hölldobler et al. 2021): Language inheritance, embedding, and aggregation. Language inheritance generally refers adopting and expanding of an existing language definition. This means that a modeling language is extended by concepts and syntactic constructs in a new grammar. In MontiCore, we generally distinguish the terms language inheritance and extension. While both are referring to the same technical realization, extension is methodically more restrictive as it considers conservation modifications only, i.e., extensions without dangerous overriding. Thus, the tooling of the original language remains reusable and extendable for language extension.

Language embedding combines the constructs of different modeling languages into a single integrated variant. Accordingly, various language components are composed, allowing syntactic constructs of different concepts to be modeled together in the same artifacts. Technically, this is achieved by extending multiple language definitions and combining their sentences in a single grammar.

Language aggregation keeps the models of different modeling languages separate but uses them in a shared context of the target domain. Thus, distinct artifacts describe various aspects of a common target. For this, abstract syntax, CoCos, and symbols of the included language spaces must be synchronized. The design patterns we discuss in this paper mainly focus on language extension or embedding. While aggregation is essential for inter-model communication of homogeneous and heterogeneous modeling languages, it is out of the scope of this contribution.

3. Design Patterns for Composition Language Engineering

One of MontiCore's key objectives is fostering compositional language design with particular regard to reusing provided tooling (generated or handwritten). Thus, we have developed several design patterns for combining existing language components, integrating handwritten artifacts, and enabling black-box reusability. As these patterns are currently applied in MontiCore, they are also reported in the corresponding handbook (Hölldobler et al. 2021). Thus, some figures might be similar as they are adopted from existing experiments.

3.1. RealThis

In software engineering, sometimes multiple individual objects should act as one for the external world, allowing black-box reusability without much configuration effort. This challenge especially arises during compositional SLE. Often, various objects (sometimes throughout different language spaces) enable modular parts of the overall functionality. Here, it is often convenient to generate independent artifacts in different generation phases. The common challenge arising from excessive incorporation of individual subcomponents is the correct delegation between them. Method calls must be delegable between arbitrary objects without establishing strong coupling through n to n relationships.

For this purpose, MontiCore uses the *realThis* pattern. It combines the composite pattern with delegation and callback, managing an object group as a single instance. Thus, generally, there is a single class whose instance acts as the externally available object. This object contains all other objects relevant for providing the composed functionalities and serves as a distributor for the incoming method calls by delegating these to the respective instances. If a sub-instance wants to call the functionality of another, it first delegates this call via a callback to the main object, the so-called `realThis`, which then forwards the call to the corresponding object.

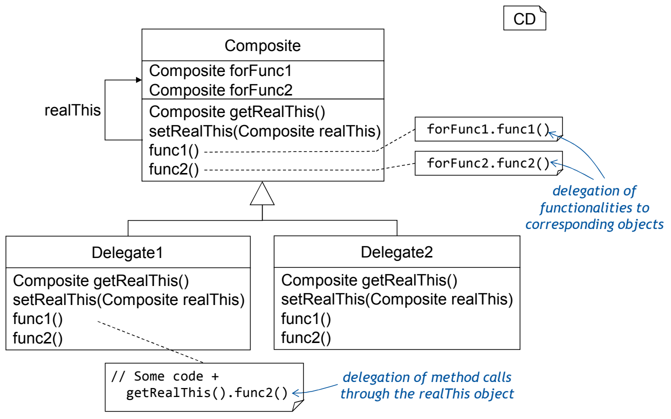


Figure 1 Example for the realThis pattern with three classes, where the `Composite` class distributes the functionalities to its attached subclasses `Delegate1` and `Delegate2`.

To allow delegation across all composed objects with integrated callback, these instances must have a shared origin, represented by the composing realThis object. All objects are instances of a subclass of the composing class and thus provide getter and setter methods for the realThis object. Figure 1 shows a class diagram with a brief example of the corresponding architecture. Class `Composite` represents the superclass of the composed objects. For each provided functionality, it contains an attribute of its own type (cf. `forFunc1` and `forFunc2`). When the corresponding methods are called, the `realThis` instance delegates to the accordingly mounted subobject. In this example, two subclasses exist, which come with their specific realizations of `func1` and `func2`. The code snippet of `Delegate1`'s `func1` shows how the objects interact with each other. The method func1 uses func2. Since the specific realization depends on the overall context (i.e., the configuration in class Composite), func2 is not called directly but indirectly via realThis.

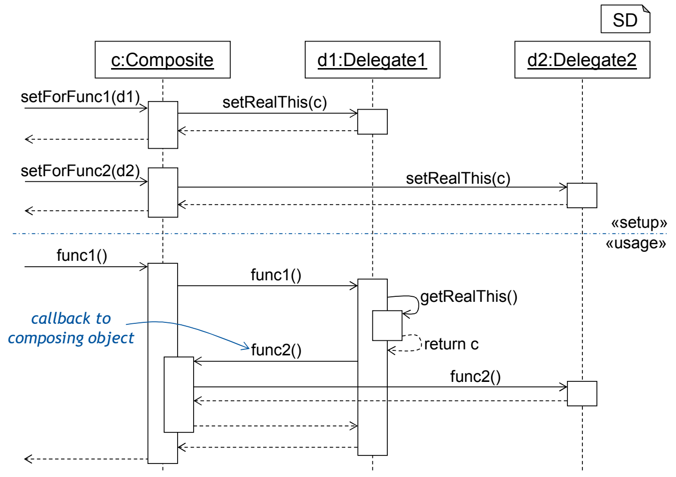


Figure 2 Sequence diagram for the execution sequence in an architecture following the realThis pattern.

Figure 2 shows the general setup and usage of the object structure. In this example, there are three objects in total: Container c:Composite, which represents the shared realThis instance, and the objects d1:Delegate1 and d1:Delegate2 for the sub-functionalities func1 and func2. First, the object group is initialized. Therefore, the sub-objects for the selected functionalities are attached to the composing object c using the corresponding setter methods (e.g., setForFunc1). To enable a callback during runtime, objects d1 and d2 have to know the realThis instance. For convenience reasons, this is performed directly from the composing object.

Using the functionalities of the assembled object group always starts from the encapsulating instance, the realThis (i.e., object c of Figure 2). In this example, we call func1. As a corresponding object is available for this functionality (i.e., d1), the call is delegated to it. Following the specification of Figure 1, d1 contains the functionality func1 but requires func2, which is provided externally in d2. Since the invocation sequence is configured via realThis, d1 performs a callback to the main object with the instruction to invoke func2. Finally, the call is delegated to d2, which completes the behavior sequence in this example.

This way, objects without any factual knowledge about each other can interact and function as a composed unit. The realThis instance is the only entity that knows the configuration of the functionalities and their associated objects. Although the structure shown appears rather shallow, the pattern can, in fact, operate on any number of indirection levels. Thus, for example, object d1 could delegate its functionality again. Important is that the actual realThis instance is passed on transitively to all contributing objects.

Generally, an attribute does not have to be set for every functionality. If it remains empty, this functionality is simply not used, which can be sufficient in various situations. On the other hand, an object can also offer several functions when hooked into the composing object multiple times.

Despite the advantages of the pattern, especially in compositional SLE, its use always comes with a certain overhead. For instance, the object group must be initialized correctly, which means that the composing objects must have their attributes assigned correctly, and the realThis instance is set properly for all objects. Although the initialization for a default configuration can often be generated, a manual setup, if required, is cumbersome. Additionally, many indirection levels lead to a huge stack of method delegations during runtime, which can affect runtime and traceability.

The described design explains the original variant of the realThis pattern. Furthermore, there are different variants with advantages and disadvantages. For example, the pattern can also be implemented without a common superclass. The generated visitor infrastructure (cf. Section 3.5) uses a variant in which classes of the composed objects have no inheritance relationship to the class of the composing object to deliberately keep inheritance hierarchies flat.

3.2. TOP Mechanism

When creating a modeling language and associated infrastructure, there eventually comes the point when the generated infrastructure needs to be extended manually. The seamless integration of handwritten into generated code poses a special challenge since, on the one hand, the generated functionality should be extended with as little effort as possible. Still, on the other hand, a new generation must not overwrite the manual modifications. Therefore, considering the separation of concerns (Hürsch & Lopes 1995), it is widely accepted that handwritten and generated artifacts should be strictly separated from each other. Please note that there are other approaches for directly integrating handwritten sources code into generated artifacts. For example, the Eclipse Modeling Framework (EMF) (Steinberg et al. 2008) allows methods to be modified directly in generated artifacts. The modifications are protected by a special tag so that they are not overwritten when the artifact is regenerated. For an extensive overview of embedding handwritten code, we refer to (Greifenberg et al. 2015).

Traditionally, the generation gap pattern (Vlissides 1998) envisions an extension of the generated artifacts via subclassing. While this strictly separates the different sources, this approach has the disadvantage of a large overhead, even for small changes. The handwritten subclasses have to be integrated at the using (or instantiating) locations of the generated code, which usually requires at least an additional customization of the corresponding builder or factory (if such design patterns are used at all).

Therefore, we have developed the TOP mechanism in MontiCore, which allows creating handwritten extensions of Java artifacts that are automatically integrated into the generated infrastructure. This mechanism has the advantage that language developers can directly extend the classes of concern without worrying about their usage. This allows for extending the implementation, such as overriding methods, changing their signatures, or introducing completely new functionality. The TOP mechanism is sensitive to handwritten artifacts given in

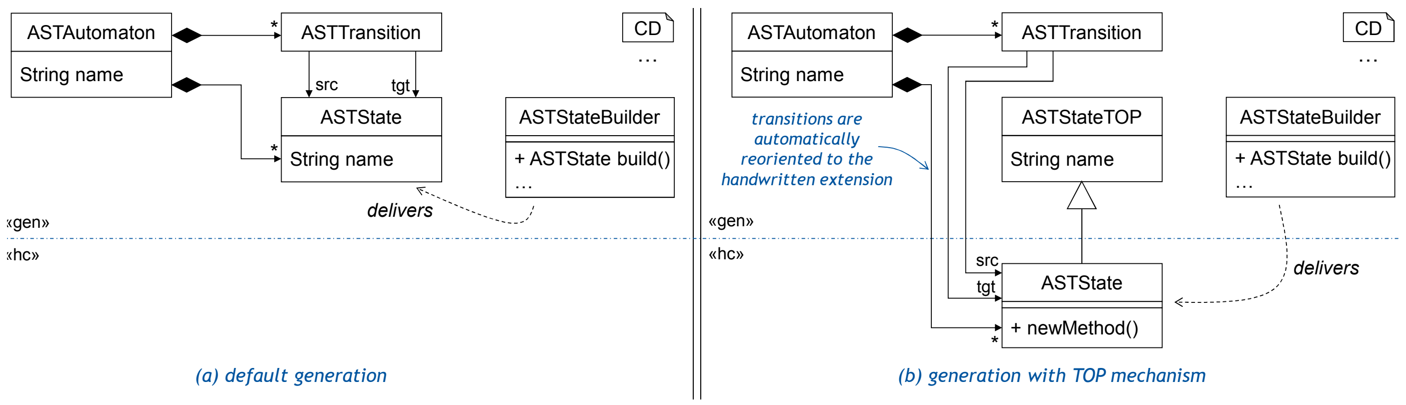


Figure 3 Application of the TOP mechanism for the generated infrastructure of an automata language. The mechanism automatically detects handwritten artifacts, creates corresponding TOP classes, and redirects associations and builder targets.

a dedicated, configurable path. For each generated class, the generator checks whether there is a corresponding hand-coded implementation available. If so, it alters the generation process such that an alternative class with the suffix `TOP`[1] is generated instead (giving this pattern its name). The TOP class features the same functionalities as the actual generated class, allowing for a comfortable extension. However, the application programming interface (API) of the remaining infrastructure (generated or handwritten) now points to the custom implementation. Thus, seamless integration is achieved without any additional effort.

We show the effect of the TOP mechanism using a simple Automaton example in Figure 3. The left-hand side (a) shows the default generation of a class `ASTAutomaton`, which contains lists of `ASTState` and `ASTTransition`. Additionally, builders are generated for all domain classes, here exemplified by the `ASTStateBuilder`. Since transitions represent connections between states, the corresponding class also has source (`src`) and target (`tgt`) associations to `ASTState`. In this situation, there are no hand-written classes yet, so the entire infrastructure is generated and self-contained.

On the right side (b), a handwritten implementation of the `ASTState` class is now added, for instance, to extend the class with new methods. Since the generator is sensitive to this class, it creates `ASTStateTOP` instead when regenerated, which contains the default implementation. The remainder of the generated infrastructure still uses the expected `ASTState` class, which means the handwritten class is already integrated without additional effort. The associations of `ASTAutomaton` and `ASTTransition` thus automatically refer to the manual implementation and no longer to the generated artifact. Furthermore, the `ASTStateBuilder` now also returns instances of the handwritten class. Since `ASTState` extends the generated artifact `ASTStateTOP`, it directly inherits all required features of the corresponding AST node.

Generally, the handwritten class does not have to inherit from the generated TOP class. However, this is strongly recommended since the generated artifacts additionally provide a lot of standard functionality, such as common interfaces or connections to the symbol table and the visitor infrastructure. Extending the generated artifacts makes the change effort minimal and applies only to the modified content.

As presented, the TOP mechanism offers a convenient solution for integrating handwritten artifacts into the generated infrastructure. It is a special variant of the generation gap pattern, which reacts sensitively to manual input and adapts the generation process accordingly. Since the customized content is seamlessly integrated, this mechanism is also particularly suitable for compositional SLE since changes are made available across language boundaries without additional effort.

3.3. Template Hook Pattern

The template hook pattern, often known as the template method in its original presentation (Gamma et al. 1995), is one of the most basic patterns in traditional object-oriented software engineering. While its application is already common practice, its fundamental concept offers particular advantages in the domain of SLE. The term *template* in this context generally means a blueprint for object-oriented behavior implementations and does not refer to FreeMarker templates that MontiCore uses for generation.

In the template hook pattern, an algorithm is implemented such that the general behavior is realized in a given blueprint method, the so-called template. Partial computations are shifted into modular functions, which are not part of the template itself. The template calls these more modular functionalities, which results in the so-called hooks. This allows customizing the general algorithm by simply overriding individual hook methods during subclassing. As a result, a developer does not always have to redevelop an entire algorithm for marginal changes, which generally facilitates the separation of concerns.

In MontiCore, we exhaustively apply the template hook pattern as it is a perfect supplement when integrating handwritten implementations (Hölldobler et al. 2021). Thus, in MontiCore, we design the generated infrastructure explicitly for providing various hook methods. In this case, such hooks lead to flexibly extensible generated code. As introduced in Section 3.2, MontiCore uses the TOP mechanism to integrate handwritten code with generated parts. As this mechanism utilizes subclassing as

[1] TOP is written in all capital letters to contrast with class names as they are rarely written in upper case.

well, it perfectly integrates with the template hook method, as it allows for overriding specific parts. Thus, almost all methods in a generated class can act as hook methods. As in the case of the TOP mechanism, the generator is sensitive to handwritten extensions when subclassing and renames the generated superclass, such that the handwritten class not only adapts the implementation but also extends the signature. This is an advantage over the traditional generation gap (Vlissides 1998) pattern as it fosters a seamless integration of handwritten artifacts in the generated infrastructure without further integration effort. This also supports compositional language design, as manually filled hooks are automatically reused in tooling across the language spaces.

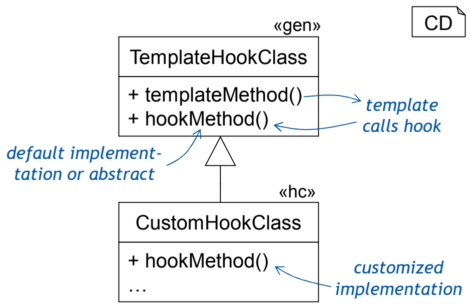


Figure 4 Template hook method via subclassing.

In general, MontiCore mainly uses two different realizations of the template hook pattern: an integrated and a delegated variant. Figure 4 shows an example of the integrated option, in which a hook method is directly overwritten via subclassing. This version is the standard form and is used, for example, when using the TOP mechanism (cf. Section 3.2). In contrast, Figure 5 shows a variant in which hook methods are relocated to a corresponding hook class. This has the effect that the template class always implements the same behavior and delegates hook calls to external classes, which can vary with the implementation. This variant contains more overhead but has the advantage that a hook's behavior is interchangeable even during the run time. In MontiCore, this variant is used, for example, in the visitor infrastructure (cf. Section 3.5).

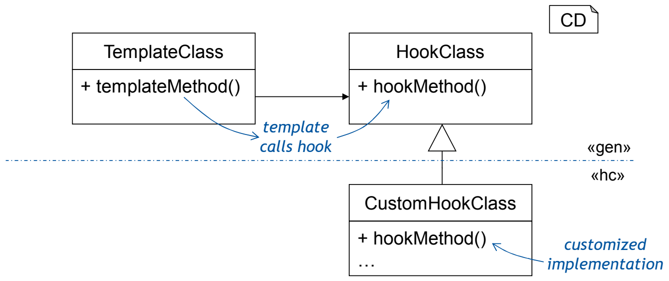
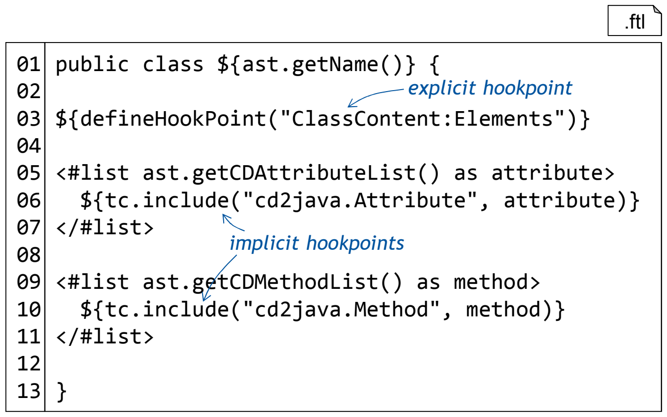


Figure 5 Template hook method using delegation to explicit hook classes

In MontiCore, we apply this design pattern not only to extend the generated infrastructure but also to extend the generator itself. MontiCore uses the FreeMarker template engine for code generation. The textual templates (not to be confused with the term template of the overall pattern) are transformed into executable Java artifacts. We extend the engine with a controller, coordinating the management FreeMarker templates. Figure 6 shows an exemplary template in FreeMarker for generating a simple class. In the body of this class, we define an explicit hook point for the manual extension of the class contents (cf. `defineHookPoint(...)`, l.3). By default, this hook point is empty and will be skipped during generation. However, if a generator user binds this hook point to one or multiple FreeMarker templates, the controller will translate these templates and integrate the result concerning the current content. Furthermore, the FreeMarker template in Figure 6 presents delegating the generation of attributes (l.6) and methods (l.10) to dedicated templates. In MontiCore, we use these standard template calls as implicit hook points. Thus, it is possible to reconfigure a template call with another to get more flexibility in the generation process.


.ftl
explicit hookpoint
implicit hookpoints

```
01 public class ${ast.getName()} {
02
03 ${defineHookPoint("ClassContent:Elements")}
04
05 <#list ast.getCDAttributeList() as attribute>
06   ${tc.include("cd2java.Attribute", attribute)}
07 </#list>
08
09 <#list ast.getCDMethodList() as method>
10   ${tc.include("cd2java.Method", method)}
11 </#list>
12
13 }
```

Figure 6 Template hook method for the FreeMarker generator engine using implicit and explicit template hooks.

3.4. Mill

Even with powerful patterns for integrating handwritten and generated artifacts or composing multiple objects, enabling black-box reuse of code poses a specific challenge in SLE. Implementations are always written against the current API, so the passed data structure of the composed language might not match the functionality's infrastructure. In fact, this problem arises for each kind of composable component. For instance, the data structure must be extensible concerning subclasses of known data types that allow the data structure's extension without changing the provided functionality of a component. Simultaneously, black-box reuse of component functionality, even on a slightly new data structure, must be preserved without requiring adaptations for the extended, composed case or any need to be recompiled. Thus, component functionality should also be usable in a compositional environment, even if it creates new objects.

This compositional setting is especially troublesome because even the mere instantiation of an object or the traversal of a composed data structure creates enormous challenges. When instantiating an object, a constructor cannot be used because the specific data type is unknown (and always different) in the compositional context. The general use of builders or factories is appropriate here, but in this case, it is only of limited use since these also have to be instantiated on the one hand but must also be kept completely interchangeable, ultimately resulting in the same issue. Accordingly, we need an additional indirection level, a kind of builder for the builder, which can be configured statically and globally from the outside for the respective composed context. We call this construct the *mill*, which gives the design pattern its name.

The basic idea of the mill is to give the developer a functionality of a static access point, which can be used to get all varying instances. Thus, for example, the mill can suitably instantiate any builder, which then, in turn, creates the actual classes. A developer does not have to care about the specific type of the builder but only asks the mill for the appropriate instance.

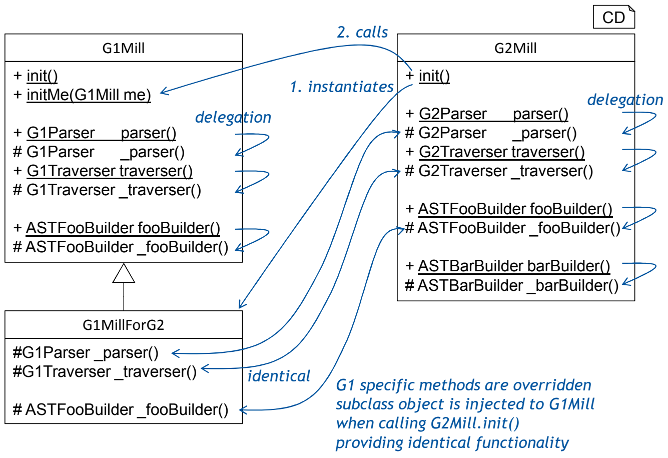

CD
G1Mill
+ init()
+ initMe(G1Mill me)
+ G1Parser parser()
# G1Parser _parser()
+ G1Traverser traverser()
# G1Traverser _traverser()
+ ASTFooBuilder fooBuilder()
# ASTFooBuilder _fooBuilder()
G1MillForG2
#G1Parser _parser()
#G1Traverser _traverser()
# ASTFooBuilder _fooBuilder()
G2Mill
+ init()
+ G2Parser parser()
# G2Parser _parser()
+ G2Traverser traverser()
# G2Traverser _traverser()
+ ASTFooBuilder fooBuilder()
# ASTFooBuilder _fooBuilder()
+ ASTBarBuilder barBuilder()
# ASTBarBuilder _barBuilder()
2. calls
1. instantiates
delegation
delegation
identical
G1 specific methods are overridden
subclass object is injected to G1Mill
when calling G2Mill.init()
providing identical functionality

Figure 7 The mill contains static and non-static methods to create a global yet configurable accessor for the infrastructure.

In MontiCore, we use this concept not only for domain classes (i.e., the AST) but also for any construct exchangeable in the compositional context, such as the parser, visitor infrastructure, or symbol table. Figure 7 shows the generated mills for two languages, `G1` and `G2`, where `G2` extends `G1`. In this example, we assume that `G1` provides the class `ASTFoo`. `G2` overrides this class with its custom variant and also introduces the class `ASTBar`. The mill itself follows the static delegator pattern. This means that for each functionality, a static method is designated as the external access point, while an internal instance method provides the actual functionality. For instance, this can be observed in `G1` since two methods are provided for each of the provider functionalities. The respective static method delegates to the instance method. When instantiating an `ASTFooBuilder`, for example, a developer uses the call of the static method `G1Mill.fooBuilder()`. It delegates to the particularly configured instance method. By default, in the non-compositional case, this is the `_fooBuilder()` method of the same class.

The mill for the `G2` language operates analogously. The interesting scenario occurs when functionalities provided at the `G1` level should be used in the compositional context of `G2`, for instance, the instantiation of the `ASTFooBuilder`. As the class `ASTFoo` is overridden in the context of `G2`, so is its builder, and thus the corresponding `G2` variant of the builder must be used. However, the mill pattern enables exactly this flexible exchangeability from the outside. Calling the `init` method of the `G2Mill` also automatically invokes the `initMe` method of the `G1Mill` with an instance of the `G1MillForG2`. This class is the link between the two language spaces and overrides the instance methods of the `G1Mill`. If a static access method is now called within the `G1Mill`, it delegates to the corresponding methods of `G1MillForG2`, which in turn provides the required functionalities from the `G2Mill` instance.

This way, functionalities of different language spaces can be developed independently but still be used collaboratively in a compositional context. It is noteworthy that while the mill pattern is very powerful, it also comes with great responsibility towards the developers. On the one hand, it is important to use the mill or a builder obtained via the mill for each instantiation. If developers do not follow this convention, it can lead to invalid instances in the data structure or, in the worst case, create type clashes at runtime for the compositional case. On the other hand, the initialization of the mill must be performed only once at the beginning of a program run. This explicitly means that a provided functionality should only use a mill but never reinitialize or reset it since this has unpredictable consequences in the composed infrastructure.

3.5. Visitor Infrastructure for Compositional Language Design

Implementing sophisticated tooling for a modeling language typically involves providing functionalities or analyses for a parsed model, requiring traversing the AST or symbol table. Following the separation of concerns, it is often advantageous to implement the operations separately from the explicit nodes and reuse a shared traversal algorithm over the model instance. The visitor pattern (Gamma et al. 1995) tackles this problem by transversing an (often tree-like) data structure and providing *visit* methods for the individual node types, which realize the desired operations. Thereby the navigation is decoupled from the actual functionality, which allows generating the general traversal strategy for a language, such that only the custom implementation has to be added manually.

MontiCore provides a customized realization of this pattern, tailored for language engineering and reusability in a compositional scenario. Here, the traversal algorithm is separated from the data structure to keep it independently adjustable. Generally, MontiCore generates the essential infrastructure for the visitor-pattern realization. For each visitable AST node (or symbol) of a language, the following methods (Hölldobler et al. 2021) are provided by default (here, on the example of the AST node `ASTFoo`):

- `handle(ASTFoo node)` defines the iteration algorithm on a `node`. By default, it calls `visit`, `traverse` (the children), and `endVisit`.
- `traverse(ASTFoo node)` defines a climbdown strategy

(i.e., the traversal of the children). The default is depth-first with no guaranteed order.

- `visit(ASTFoo node)` is called when entering the `node`. By default, it contains no behavior and represents a customizable extension point for the desired functionality.

- `endVisit(ASTFoo node)` is called when leaving the `node`. It is an analog hook point to the `visit` method.

Furthermore, each traversable node implements an `accept` method. The visitor infrastructure triggers this method to simulate double dispatching. Since Java only supports single dispatching, this detour guarantees that a visited node is recognized in a type-correct manner. This is achieved by calling `accept` when traversing a node, which, in turn, triggers the `handle` method with itself as an argument.

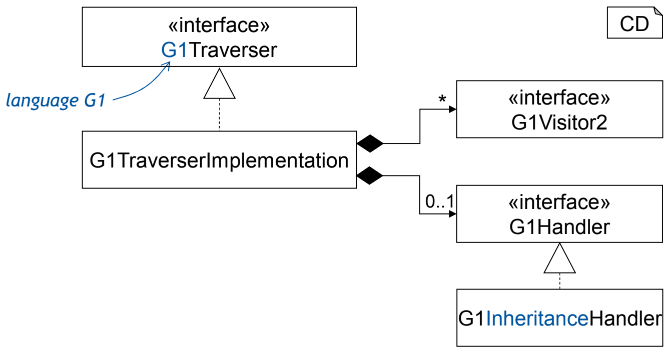


Figure 8 Generated visitor infrastructure for a language `G1`.

These methods are distributed across the generated visitor infrastructure, which includes, for each language, the interfaces `Traverser`, `Visitor2`, and `Handler`, as well as the class `TraverserImplementation`. Figure 8 presents this infrastructure for the language G1, containing the corresponding interfaces and classes, each individualized with the prefix of the respective language name.

The infrastructure follows a compositional design in which the traverser encapsulates all other objects involved in a functionality. Therefore, a variant of the realThis pattern (cf. Section 3.1) is applied, where the traverser is the conceptual access point for all actions, which delegates method calls to the correct instances. The attached objects, in turn, know the encapsulating traverser and can delegate back to it if necessary. The traverser contains the default behavior for dealing with traversed nodes, thus implementing the `handle` and `traverse` methods for all types of nodes. It applies a depth-first climbdown strategy. To customize the handle or traverse implementations, a corresponding handler can be attached. The traverser does not use its default in such a case but delegates to the new behavior. For `visit` and `endVisit` calls, it always delegates to the corresponding hooked `Visitor2` instances. The traverser is, in its realization, divided into two parts: an interface and an implementing class. Here, a language developer implements against the interface, which provides the necessary method signatures. The class additionally handles the attributes management for attached visitors or handlers (Hölldobler et al. 2021). The reason for this bipartition is the reusability of standard implementations in compositional language construction. Generally, the traverser interface extends all traverser interfaces of the extended languages, reusing their behavior. This leads to multiple inheritance, which is only possible in Java via interfaces.

The `Visitor2` interface of a language provides `visit` and `endVisit` methods for each node, intended for customization. A language developer can implement and attach these interfaces to a traverser. During traversal, the traverser delegates to the corresponding custom implementation. When entering a node, the `visit` method is called. Analogously, the `endVisit` method is triggered when leaving the respective node (i.e., when all child nodes have been traversed). In contrast to the traverser, a visitor is language-specific, containing only `visit` and `endVisit` methods for the nodes introduced in the respective grammar. Thus, the traverser supports binding visitors of all included sublanguages to implement behavior specific to different language components in a composed scenario. Furthermore, a traverser manages lists of visitors, allowing for executing multiple functionalities in a single run. When processing a node, it calls the `visit`/`endVisit` methods of all hooked visitors. While this mechanism increases efficiency, a language developer must ensure that the attached visitors do not cause any conflicts due to potential side effects.

Furthermore, MontiCore generates the `Handler` interface. While the traverser already contains a default implementation for `handle` and `traverse` methods, a handler enables their customization. It allows adjusting the traversal algorithm for specific nodes while preserving the default for the others. Similar to the visitor interface, the handler is language-specific, thus containing methods only for the associated sublanguage. The difference is that a traverser allows at most one handler for each sublanguage instead of a list since it only makes sense to adjust the traversal strategy once. According to the used variant of the realThis pattern, a handler delegates all method calls back to the traverser, which in turn manages their distribution. Thus, the handler offers to customize the traversal strategy individually. In most cases, however, the default depth-first approach should be sufficient, such that a handler is not required.

Finally, MontiCore already generates a variant of the handler, the `InheritanceHandler`. It implements the corresponding interface and overwrites its methods. The reason is that the default traversal algorithm only visits a node as its most specific type. However, since AST nodes can extend others, it is often desirable for a node to also be handled as its more general type. This is especially important in the compositional case to enable the reuse of already implemented behavior. The `InheritanceHandler` solves this problem by triggering for each node not only the most specific `visit` and `endVisit` methods but also those of its supertypes. When realizing functionalities via the visitor pattern, the `InheritanceHandler` should thus be preferred to support reusability in language composition.

Figure 9 shows the control flow of a traverser with an attached visitor implementation, here on the example of a pretty printer. The traversal is started by calling `accept(t)` on a node (here: `ast`). This, in turn, calls the `handle` method of the

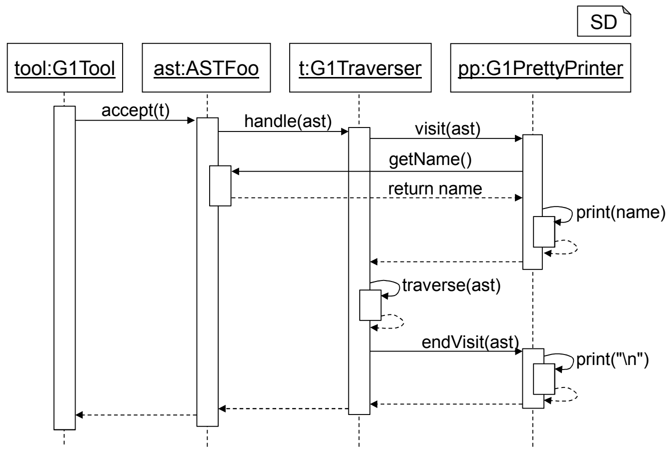
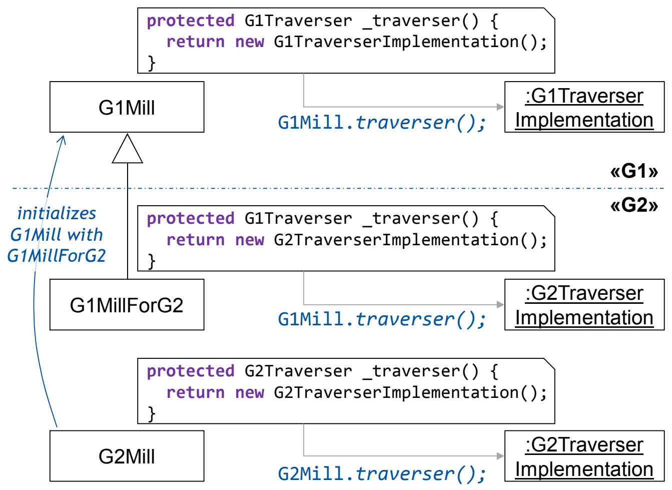


Figure 9 Control flow of a traverser with an attached visitor implementation on the example of a pretty printer.

traverser with itself as parameter, thus enabling double dispatching. The traverser delegates to the `visit` method of the pretty printer and passes `ast` as argument. The printer, in turn, reads the `name` attribute of the passed node and prints it. After completing the `visit` call, the traverser traverses the node. In this simple example, we assume that there are no other child nodes, leaving this operation empty. Thus, the traverser finally triggers the printer's `endVisit` method, which produces a line break in the output. This way, the presented infrastructure enables performing complex operations on AST and symbol table. The complexity is largely hidden to the language developer, who ideally only has to implement visitor interfaces and hook these into a traverser.



```
protected G1Traverser _traverser() {
  return new G1TraverserImplementation();
}
```

```
G1Mill.traverser();
```

```
protected G1Traverser _traverser() {
  return new G2TraverserImplementation();
}
```

```
G1Mill.traverser();
```

```
protected G2Traverser _traverser() {
  return new G2TraverserImplementation();
}
```

```
G2Mill.traverser();
```

G1Mill
:G1Traverser Implementation
«G1»
«G2»
initializes G1Mill with G1MillForG2
G1MillForG2
:G2Traverser Implementation
G2Mill
:G2Traverser Implementation

Figure 10 The instantiation of a traverser via mill in a composed scenario. The mill always returns the correct traverser instance with respect to the current language space.

The presented visitor infrastructure is tailored explicitly for its application in compositional language engineering. As explained, the traverser interfaces inherit from their respective sublanguage counterparts to provide the appropriate methods for the nodes of all included language components. Nevertheless, a developer can only implement functionality based on the traverser known in the current language. Overall, this limits the black-box reusability of visitor configurations since an extending language may introduce new node types that are unknown to the original traverser. To enable reusability without manual adjustments, the traverser must be automatically replaced by a more specific variant. MontiCore realizes this functionality in combination with the mill pattern. Figure 10 shows the adaptive replacement of the traverser using the example introduced in Section 3.4. The language `G2` extends `G1`. To ensure a correct traverser instance at each language level, the traverser must always be initialized via the associated mill. Thus, a developer for `G1` functionalities must consistently create it using the mill with the static call `G1Mill.traverser()`. Due to the realization of the mill as static delegator, the same call leads to a corresponding instance of the `G2Traverser` in the context of `G2`. This is because the actual instance of `G1Mill` has been replaced by `G1MillForG2`, which delegates the instantiation. Thus, by combining the mill pattern with the concrete realization of the visitor pattern, MontiCore supports adaptable traversers enabling black-box reusability in language composition.

4. Discussion

The presented design patterns for compositional SLE are based on our experiences in engineering modeling languages in MontiCore. However, the catalog is obviously just a starting point and does not claim to be complete. Other patterns are conceivable, which are tailored to specific issues. Table 1 provides a condensed overview of the presented catalog. Generally, applying these patterns enhances functionality, especially for language composition, but often increases complexity and requires additional attention by language developers.

The presented patterns have been developed to support the engineering of modeling languages, especially in a compositional context, in which the syntax and existing tooling of the individual language spaces can be reused. However, despite applying these patterns, it is still possible to build invalid composed languages. Constructs of different language components may be mutually exclusive, or a language designer may explicitly choose to override pre-existing productions restricting the overall language. This could cause parsing issues but also conflicts and clashes in the AST. To improve this situation, we introduce the term conservative extension in MontiCore (Hölldobler et al. 2021). Languages are extended conservatively if all models of the extended sub-languages are still valid models of the resulting composed language. This implies fewer conflicts (such as type clashes) and increased black-box reusability of existing tooling on the AST.

While the patterns provide substantial support for compositional SLE, language engineers also require a large amount of foresight to adhere to a pattern's specifications. This is most obvious in the example of the mill (cf. Section 3.4), which must be used consistently for instantiating new objects and must not be initialized or reset negligently. In this case, the application is particularly deceptive since engineers have hardly any im-

Table 1 Overview of the proposed design patterns for compositional SLE with use case, objectives, and (dis-) advantages.

realThis	
Use Case	Multiple individual objects should act as a single instance for the external world.
Objectives	• combines composite with delegation and callback
	• manages an object group as a single instance
Advantages	• black-box reusability of composed functionalities
	• shared central entity reduces number of relationships
	• generation of default configurations possible
Disadvantages	• many delegation steps during runtime
	• custom configurations potentially cumbersome
TOP mechanism	
Use Case	Automatic integration of handwritten artifacts into the generated infrastructure.
Objectives	• detects custom counterparts of generated artifacts
	• incorporates handwritten classes instead of generated
Advantages	• seamless integration of handwritten code
	• no additional integration effort
	• no overriding of customizations when re-generating
Disadvantages	• naming convention
template hook	
Use Case	Splitting a computation or generation step into multiple modular functionalities.
Objectives	• enables exchangeable atomic functionalities
	⇒ facilitates the integration of handwritten code
	• modularization of code generators
Advantages	• small modification and extension effort
	• good cooperation with the TOP mechanism
	• modular computation steps and generation templates
Disadvantages	• additional maintenance effort for template entities
mill	
Use Case	For composed languages, objects should be instantiated adaptively with respect to their current context.
Objectives	• provides a globally exchangeable *builder for builders*
Advantages	• black-box reusability of compsed functionalities
	• flexible exchange of builders due to a static delegator
Disadvantages	• direct benefits hidden for base language developers
	• responsibility to intrinsically adhere to the pattern
visitor infrastructure for compositional SLE	
Use Case	Implementing additional functionalities on a (potentially compositional) data structure
Objectives	• traverses data structure to add external functionalities
Advantages	• adding functionality without altering of data structure
	• automatic adaption of traversal to new data structure
	• black-box reusability of composed functionalities
	• built-in double-dispatching
Disadvantages	• rather complex compared to original visitor pattern

mediate advantages within their own language compared to the direct use of corresponding builders. The effects of the pattern, and thus also potential errors due to incorrect use, only become apparent during composition and the intended black-box reuse of functionalities.

Our generated infrastructure is designed to be particularly robust for composed SLE while adhering to the presented design patterns. However, this comes with the disadvantage of an excessively complex structure, increasing the barrier to entry for development. For instance, the realization of the visitor pattern splits the functionality across multiple composed elements. While this increases reusability, it imposes a rather complex design compared to the original visitor pattern. Nevertheless, we think this complexity is justified as the standard case is still relatively straightforward, and the complexity ultimately pays off for sophisticated composed modeling languages.

Finally, our work concentrated on design patterns for SLE that could foster reusability in a compositional setting. However, we did not elaborate on the composition of their code generators. Generator composition is an interesting field with open challenges, such as synchronizing existing generators or their produced artifacts. While some application domains have specifically tailored approaches (Ringert et al. 2015), the overall topic is still ongoing research.

5. Related Work

Different software engineering design patterns have been presented in detail over the past decades (Gamma et al. 1995). They help to develop, maintain, and sustainably evolve modern software systems. In the following, we provide an overview of patterns related to our catalog. Often, these are related to existing ones or introduce variations for compositional SLE.

5.1. Delegation and State

Generally, the behavior of a given class is modifiable using inheritance. Any call to the overridden method is handled by the overriding method. An alternative to behavior modification in object-oriented programming languages is given by *delegation* (Lieberman 1986) (also known as *redirection* (Smith & Stotts 2002)) based on object composition.

During delegation, the object receiving the request delegates the request to a contained object (*delegate*) which handles the request. The object's behavior can thus be modified by exchanging the delegate. Additionally, multiple methods using multiple delegates allow for more flexibility than given by inheritance, especially for programming languages only allowing for single inheritance, such as Java.

Using a single delegate in which all requests are passed toward one can alter an object's behavior by exchanging the delegate in a similar way to interchanging the object with another object where both objects' classes differ but derive from the same base class. However, unlike an actual change of the object, the *state pattern* allows to seemingly change the state of the object without updating all of its references. We use the additional flexibility provided by the state pattern in the realThis pattern, where we, instead of having multiple states, store multiple delegates at the same time.

5.2. Strategy

Interchangeability of algorithms can be achieved by having each algorithm contained in its particular class (*concrete strategy*) (Gamma et al. 1995). Each concrete strategy implements a common interface (*strategy*) that can be used in a *context*. Interchanging a concrete strategy with another is reflected in a change of the program's behavior when the strategy is called. This pattern can be used in conjunction with delegation, changing the object's behavior. Alternatively, a strategy is transmittable to and from methods enabling complex algorithm design. Overall, the strategy pattern is fairly comparable to the state pattern. In the same way, the realThis pattern may be used to combine strategies using multiple partial implementations.

5.3. Template Method

When encountering multiple method implementations that only partially differ, decreasing code duplication can be achieved by exchanging these methods with one template method (Gamma et al. 1995). The template method contains the common parts of the given algorithms, while the the original parts are given by calls to *hook points*; These hook point methods (re-)define the details of the algorithms. The behavior of the template method can be changed by overriding the hook point methods in a subclass. Alternatively, the template method pattern can be combined with delegation, implementing the original parts in sub-classes of the delegates. The TOP mechanism makes heavy use of this concept by having every method in the generated code be a hook point with default behavior to potentially be overridden in the corresponding subclass.

5.4. Bridge

As the implementation of interfaces by inheritance strongly couples the implementation to the respective interfaces, further decoupling enables greater future modifiability. To this extent, an interface and its implementations are replaced by a class delegating its request, forming a *bridge* between interface and implementation (Gamma et al. 1995). A change of the delegate does not require changes in the derived classes of the class responsible for delegating. Replacing the delegate has a similar effect as replacing the base class, which implements the interface without modification to the derived class. This concept is compatible with the realThis pattern, as any class containing multiple delegates can be further extended with more methods and their implementations can again be realized using the realThis pattern.

5.5. Adapter

Another option to decouple interfaces and their implementations is not requiring the implementations to implement the specific interfaces. Instead, an *adapter* class is introduced, which implements the interface (Gamma et al. 1995). The adapter then uses the class with a different interface (*adaptee*) to implement the target interface. The adaptee only consists of lightweight logic to mitigate the difference between the adaptee's interface and the target one. The adapter pattern allows to implement classes and use them without modifications in a new context. Figure 7 shows how this is used in the mill pattern. `G1MillForG2` implements the interface of `G1Mill` using the implementations provided by the `G2Mill`.

5.6. Factory Method

While the program's behavior can be modified using subclassing, objects of said subclasses are required and thus are required to be instantiated. As such, dependent on the given state of the program, different subclasses are required. To decouple the decision of which class to instantiate, one may use a *factory method* (Gamma et al. 1995). A factory method is used instead of a specific constructor; it returns a new instance of a corresponding subclass, while the decision of which subclass to instantiate is left to the implementation of the factory method. The factory method's behavior can itself be modified using subclassing. Thus object creation of specific subclasses can be achieved within code without knowledge of said subclasses. Factory methods are part of the mill pattern to provide the correct types when one language extends another.

5.7. Singleton

A *singleton* class is one that guarantees never to have multiple instances. Additionally, the instance is globally accessible. To this end, a static method can be used to get the only instance stored as a global state, e.g., as a static attribute of the singleton class. To control the number of instances, the constructor is made inaccessible. In its place, a (static) method may be used for initialization if and only if there is no instance already. Mill instances of the corresponding pattern are conceptually similar to singletons providing static methods with interchangeable implementations (e.g., factory methods that return different instances given language extension).

6. Conclusion and Dedication to Antonio's Birthday

Historically, we were interested in providing tools for effective agile use of the UML in software development. Discovering that developing such a tool is itself a complex task, which must become more agile and effective. In doing so, we started our efforts in developing the MontiCore language workbench, which meanwhile allows us to not only develop UML analysis and synthesis tools but also address the SysML and various domain-specific languages, where quite a number of them are not especially dedicated to software and not even to systems engineering. Agility largely comes from the ability to reuse predefined components without having to modify their internals. MontiCore, therefore, did put considerable emphasis on compositionality. The presented design patterns were to some extent developed, adapted, or also reused directly for this effort.

Much of the work done in MontiCore was heavily influenced by Antonio's work. For instance, Antonio's work on UML profiles (Fuentes-Fernández & Vallecillo-Moreno 2004) inspired us to look at customizability and especially variability of languages (Butting et al. 2018, 2021), both on semantics and syntactic elements. A special highlight also was his works on using functional programs and term rewriting techniques available in Maude to breathe behavioral life into the UML (Romero et al. 2007; Rivera et al. 2009).

His and his colleagues' work on model transformations (Vallecillo et al. 2012; Troya Castilla & Vallecillo Moreno 2011; Rivera et al. 2009; Gogolla & Vallecillo 2011) helped us to understand how to improve internal transformations when synthesizing code, but also what is needed to systematically synthesize explicit transformation languages that are based on the concrete syntax of the underlying modeling language (Hölldobler et al. 2015).

The WebML approach (Moreno et al. 2007) showed us how to use the UML for modeling and cutting out larger parts of a complete software system as done in MontiGem (Gerasimov, Michael, et al. 2020) for generating financial management systems (Gerasimov, Heuser, et al. 2020), digital twin cockpits (Dalibor et al. 2020), low-code platforms (Dalibor et al. 2022), or assistive systems (Michael et al. 2020).

There are many more such influences, e.g., Antonio's work on model differencing (Rivera & Vallecillo 2008) that we use for our model differencing techniques (Maoz et al. 2011).

This all shows that Antonio is a brilliant and diligent software engineer who is an important member of our community with many, many highly valuable contributions. Antonio is and has always been a very inspiring colleague when we met in various conferences and workshops over the years.

Our work would not have been the same if he wasn't there.

References

Blom, H., Lönn, H., Hagl, F., Papadopoulos, Y., Reiser, M.-O., Sjöstedt, C.-J., … others (2013). EAST-ADL: An architecture description language for Automotive Software-Intensive Systems. *Embedded Computing Systems: Applications, Optimization, and Advanced Design: Applications, Optimization, and Advanced Design*, 456.

Bryant, B., Jézéquel, J.-M., Lämmel, R., Mernik, M., Schindler, M., Steinmann, F., … Völter, M. (2015). Globalized Domain Specific Language Engineering. In B. Combemale, B. H. Cheng, R. B. France, J.-M. Jézéquel, & B. Rumpe (Eds.), *Globalizing Domain-Specific Languages* (p. 43-69). Springer. doi: 10.1007/978-3-319-26172-0_4

Butting, A., Eikermann, R., Hölldobler, K., Jansen, N., Rumpe, B., & Wortmann, A. (2020, October). A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Special Issue dedicated to Martin Gogolla on his 65th Birthday, Journal of Object Technology*, *19*(3), 3:1-16. (Special Issue dedicated to Martin Gogolla on his 65th Birthday)

Butting, A., Eikermann, R., Kautz, O., Rumpe, B., & Wortmann, A. (2018, September). Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM.

Butting, A., Hölldobler, K., Rumpe, B., & Wortmann, A. (2021, July). Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques - The MontiCore Approach. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen (Ed.), *Composing Model-Based Analysis Tools* (p. 217-234). Springer.

Butting, A., Rumpe, B., & Wortmann, A. (2016, October). Embedding Component Behavior DSLs into the MontiArcAutomaton ADL. In *Globalization of Modeling Languages Workshop (GEMOC'16)* (Vol. 1731).

Cengarle, M. V., Grönniger, H., & Rumpe, B. (2009). Variability within Modeling Language Definitions. In *Conference on model driven engineering languages and systems (models'09)* (p. 670-684). Springer.

Clark, T., Brand, M. v. d., Combemale, B., & Rumpe, B. (2015). Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages* (p. 7-20). Springer.

Dalibor, M., Heithoff, M., Michael, J., Netz, L., Pfeiffer, J., Rumpe, B., … Wortmann, A. (2022). Generating Customized Low-Code Development Platforms for Digital Twins. *Journal of Comp. Lang. (COLA)*, *70*. doi: 10.1016/j.cola.2022.101117

Dalibor, M., Michael, J., Rumpe, B., Varga, S., & Wortmann, A. (2020, October). Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In G. Dobbie, U. Frank, G. Kappel, S. W. Liddle, & H. C. Mayr (Eds.), *Conceptual Modeling* (p. 377-387). Springer International Publishing.

Erdweg, S., Storm, T. v. d., Völter, M., Boersma, M., Bosman, R., Cook, W. R., … others (2013). The State of the Art in Language Workbenches. In *International Conference on Software Language Engineering* (pp. 197–217).

Favre, J.-M. (2005). Languages evolve too! Changing the Software Time Scale. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)* (pp. 33–42).

Feiler, P. H., & Gluch, D. P. (2012). *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley.

France, R., & Rumpe, B. (2007, May). Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)*, 37-54.

FreeMarker website. (2022). https://freemarker.apache.org/. (Accessed: 2022-05-10)

Friedenthal, S., Moore, A., & Steiner, R. (2014). *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann.

Fuentes-Fernández, L., & Vallecillo-Moreno, A. (2004). An introduction to UML profiles. *UPGRADE, European Journal for the Informatics Professional*, *2*.

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

Gerasimov, A., Heuser, P., Ketteniß, H., Letmathe, P., Michael, J., Netz, L., … Varga, S. (2020, February). Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In J. Michael & D. Bork (Eds.), *Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers* (p. 22-30).

CEUR Workshop Proceedings.

Gerasimov, A., Michael, J., Netz, L., Rumpe, B., & Varga, S. (2020, August). Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In B. Anderson, J. Thatcher, & R. Meservy (Eds.), *25th Americas Conference on Information Systems (AMCIS 2020)* (p. 1-10). Association for Information Systems (AIS).

Gogolla, M., & Vallecillo, A. (2011). Tractable Model Transformation Testing. In *European Conference on Modelling Foundations and Applications* (pp. 221–235).

Greifenberg, T., Hölldobler, K., Kolassa, C., Look, M., Mir Seyed Nazari, P., Müller, K., ... Wortmann, A. (2015). Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development* (Vol. 580, p. 112-132). Springer.

Harel, D., & Rumpe, B. (2004, October). Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, *37*(10), 64-72.

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

Hölldobler, K., Rumpe, B., & Weisemöller, I. (2015). Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)* (p. 136-145). ACM/IEEE.

Hölldobler, K., Rumpe, B., & Wortmann, A. (2018). Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures*, *54*, 386-405.

Hürsch, W. L., & Lopes, C. V. (1995). Separation of Concerns.

Kleppe, A. (2008). *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Pearson Education.

Klint, P., Lämmel, R., & Verhoef, C. (2005). Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *14*(3), 331–380.

Kurtev, I., Bézivin, J., & Aksit, M. (2002). Technological Spaces: An Initial Appraisal. *CoopIS, DOA*, *2002*.

Lieberman, H. (1986). Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Conference proceedings on Object-oriented programming systems, languages and applications* (pp. 214–223).

Maoz, S., Ringert, J. O., & Rumpe, B. (2011, October). Summarizing Semantic Model Differences. In B. Schätz, D. Deridder, A. Pierantonio, J. Sprinkle, & D. Tamzalit (Eds.), *Me 2011 - models and evolution.*

Medvidovic, N., & Taylor, R. N. (2000). A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, *26*(1), 70–93.

Michael, J., Rumpe, B., & Varga, S. (2020, June). Human Behavior, Goals and Model-Driven Software Engineering for Assistive Systems. In A. Koschmider, J. Michael, & B. Thalheim (Eds.), *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)* (Vol. 2628, p. 11-18). CEUR Workshop Proceedings.

Moreno, N., Fraternali, P., & Vallecillo, A. (2007). WebML modelling in UML. *IET Software*, *1*(3), 67–80.

Ringert, J. O., Roth, A., Rumpe, B., & Wortmann, A. (2015). Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, *6*(1), 33-57.

Rivera, J. E., Durán, F., & Vallecillo, A. (2009). Formal Specification and Analysis of Domain Specific Models Using Maude. *SIMULATION*, *85*(11-12), 778-792. doi: 10.1177/0037549709341635

Rivera, J. E., & Vallecillo, A. (2008). Representing and Operating with Model Differences. In *International Conference on Objects, Components, Models and Patterns* (pp. 141–160).

Romero, J. R., Rivera, J. E., Durán, F., & Vallecillo, A. (2007). Formal and Tool Support for Model Driven Engineering with Maude. *J. Object Technol.*, *6*(9), 187–207.

Rumpe, B. (2016). *Modeling with UML: Language, Concepts, Methods*. Springer International.

Selic, B. (2003). The Pragmatics of Model-Driven Development. *IEEE software*, *20*(5), 19–25.

Smith, J., & Stotts, D. (2002). An Elemental Design Pattern Catalog. *Technical Report TR-02–040*.

Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *EMF: Eclipse Modeling Framework*. Pearson Education.

Troya Castilla, J., & Vallecillo Moreno, A. (2011). A Rewriting Logic Semantics for ATL. *Journal of Object Technology, 10, 5: 1-5: 29*.

Vallecillo, A. (2010). On the Combination of Domain Specific Modeling Languages. In T. Kühne, B. Selic, M.-P. Gervais, & F. Terrier (Eds.), *Modelling Foundations and Applications* (pp. 305–320). Springer.

Vallecillo, A., Gogolla, M., Burgueno, L., Wimmer, M., & Hamann, L. (2012). Formal Specification and Testing of Model Transformations. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems* (pp. 399–437).

Vlissides, J. (1998). *Pattern Hatching: Design Patterns Applied*. Addison-Wesley Longman Ltd.

Völter, M., Stahl, T., Bettin, J., Haase, A., & Helsen, S. (2013). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

Wigand, D. L., Nordmann, A., Dehio, N., Mistry, M., & Wrede, S. (2017). Domain-Specific Language Modularization Scheme Applied to a Multi-Arm Robotics Use-Case. *Journal of Software Engineering for Robotics*.

Wirth, N. (1996). Extended Backus-Naur Form (EBNF). *ISO/IEC*, *14977*(2996), 2–21.