Identifying Code Generation Candidates Using Software Categories

Pedram Mir Seyed Nazari(✉) and Bernhard Rumpe

Software Engineering, RWTH Aachen University, Aachen, Germany
nazari@se-rwth.de
http://www.se-rwth.de

Abstract. Code generators are a crucial part of the model-driven development (MDD) approach. They systematically transform abstract models to concrete executable source code. Typically, a generator developer determines the code parts that should be generated and separates them from handwritten code. Since performed manually, this task often is time-consuming, labor-intensive, difficult to maintain and may produce more code than necessary. This paper presents an iterative approach for identifying candidates for generated code by analyzing the dependencies of categorized code parts. Dependency rules are automatically derived from a predefined software category graph and serve as basis for the categorization process. Generator developers can use this approach to systematically identify code generation candidates. The ideas and concepts of this paper were introduced at the MODELSWARD conference [1] and are extended in this contribution.

Keywords: Model-Driven Development · Generators · Software categories

1 Introduction

Models are at the center of the model-driven development (MDD) [2] approach. They abstract from technical details [3], facilitating a more problem-oriented development of software. In contrast to conventional general-purpose languages (GPL, such as Java or C), the language of models is limited to concepts of a specific domain, namely, a domain-specific language (DSL). To obtain an executable software application, code generators systematically transform the abstract models to instances of a GPL [4] (e.g., classes of Java). However, code generators are software themselves and need to be developed as well. There are different development processes for code generators. One that is often suggested (e.g., [5,6]) is shown in Fig. 1.

The approach includes four steps. First, a reference model is created, which ultimately serves as input for the generator. Depending on this reference model, the generator developer (resp. tool developer [7]) creates the reference implementation. Next, it has to be determined which code parts need to be or can

© Springer International Publishing Switzerland 2015
P. Desfray et al. (Eds.): MODELSWARD 2015, CCIS 580, pp. 356–372, 2015.
DOI: 10.1007/978-3-319-27869-8_

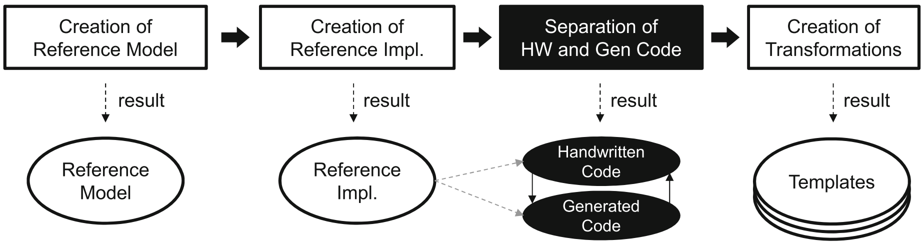

Creation of Reference Model
Creation of Reference Impl.
Separation of HW and Gen Code
Creation of Transformations
result
result
result
result
Reference Model
Reference Impl.
Handwritten Code
Generated Code
Templates

Fig. 1. Typical development steps of a code generator.

be generated and which ones should remain handwritten. Finally, the transformations are defined to transform the reference model to the aforementioned generated code.

Often, the third step, i.e., 'separation of hand-written and generated code' is not explicitly mentioned in the literature. This separation is implicit part of the last step, i.e., 'creation of transformations', since the transformations are only created for code that ought to be generated. However, the separation of handwritten and generated code ought to be distinguished as a step on its own, since it is not always obvious which classes need to be generated.

In general, every class can be generated, especially when using template-based generators. In an extreme case, a class can be fully copied into a template containing only static template code (and, thus, is independent of the input model). This is not desired, following the guideline that only as much code should be generated as necessary [5,8,9]. Optimally, most code is put into the domain framework (or domain platform), increasing the understandability and maintainability of the software. The generated code then only configures the domain framework for specific purposes [10] through mechanisms, e.g., as described in [11].

One important criterion for a code generator to be reasonable is the existence of similar code parts, either in the same software product or in different products (e.g., software product lines [12,13]). Typically, generation candidates are similar code parts that are also related to the domain. For example, in a domain about cars, the classes `Wheel` and `Brake` would be more likely generation candidates than the domain independent and thus unchanged class `File`. This, of course, is the case, since the information for the generated code is obtained by the input model which, in turn, is an instance of a DSL that by definition describes elements of a specific domain. Of course, the logical relation to the domain is not a necessary criterion, because if the DSL is not expressive enough, the generated code is additionally integrated with handwritten code. Nevertheless, the generated code often has some bearing on the domain.

In most cases, the generator developer manually separates handwritten code from generated code. This process can be time-consuming, labor-intensive and may impede maintenance. Furthermore, when using a domain framework, this separation is insufficient, since the handwritten code needs to be separated into handwritten code for a *specific project* and handwritten code concerning the *whole domain*. This separation also impacts the maintenance of the software [8]. To address this problem, software categories, as presented in [14], are suited.

The aim of this paper is to show how software categories can be exploited to categorize semi-automatically classes and interfaces of an object-oriented software system. The resulting categorization can be used for determining candidates for generated code, supporting the developer performing this separation task. The ideas and concepts were introduced at the MODELSWARD conference [1] and are extended in this contribution.

This paper is structured as follows: Sect. 2 introduces software categories and the used terminology. In Sect. 3, these software categories are adjusted for generative software. Section 4 presents the allowed dependencies derived by the previously defined software categories. The general categorization approach is explained in Sect. 5 and exemplified in Sect. 6. Section 7 outlines further possible dependencies. Concepts related to our work are discussed in Sect. 8. Finally, Sect. 9 concludes the paper.

2 Software Categories

Software systems, especially larger ones, consist of a number of components that interact with each other. The components usually belong to different kinds of categories, such as *persistence*, *gui* and *application*. Therefore, [14] suggests using software categories for finding appropriate components. In the following this idea is demonstrated by an example.[1]

Suppose that a software system for the card game Sheepshead[2] should be developed. The following categories then could be created (see Fig. 2):

- *0* (Zero): contains only global software that is well-tested, e.g., `java.lang` and `java.util` of the JDK.
- *CardGame:* contains fundamental knowledge about card games in general. Hence, it can be used for different card games.
- *SheepsHead:* Contains rules for the Sheepshead game, e.g., whether a card can be drawn.
- *CardGameGUI:* determines the design of the card game, independent of the used library, e.g., that the cards should be in the middle of the screen.
- *CardGameGUISwing:* extends Swing by illustration facilities for cards.
- *Swing:* contains fundamental knowledge about Java Swing.

[1] The example is taken from [14] and reduced to only the aspects required to explain our approach.

[2] This game (in German called *Schafkopf* or *Schaffkopf*) is a popular and complex Bavarian card game with thirty-two cards where four players play in dynamic alliances. The English translation *Sheepshead* is actually wrong as it comes from ancient times where the game was played on top (*kopf*) of a barrel (*schaff*). It has nothing to do with sheep.

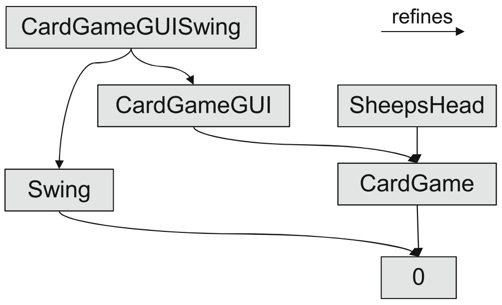


Fig. 2. Software categories for virtual *SheepsHead* [14] (shortened).

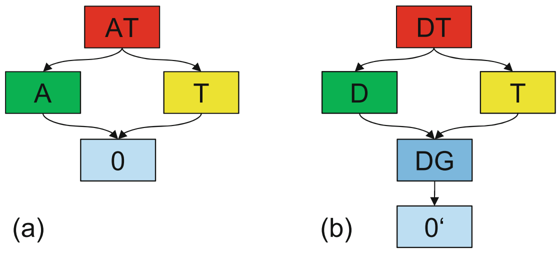


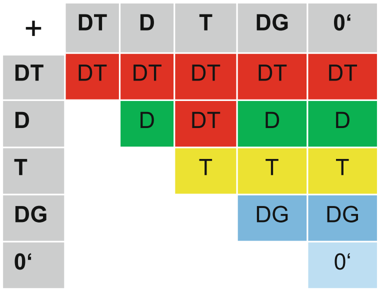
Fig. 3. Software categories (a) in general [14] and (b) adjusted for generative software.

An arrow in Fig. 2 represents a refinement relation between two categories. Classes that are in a category *C1* that refines another category *C2* may use classes of this category *C2*. The other way around is not allowed. Every category - directly or indirectly - refines the category *0* (arrows in Fig. 2). Hence, software in *0* can be used in every category without any problems. *CardGame* is refined by *SheepsHead* and *CardGameGUI* which means that code in these categories can also use code in *CardGame*. Note that a communication between *CardGameGUI* and *SheepsHead* is not allowed directly, but rather by using *CardGame* or *0* interfaces. Since the category *CardGameGUISwing* refines both *CardGameGUI* and *Swing*, it is a mixed form of these two categories.

Now, having these categories, appropriate components can be found. For example, a component `SheepsHeadRules` in the category *SheepsHead*, `CardGameInfo` and `VirtualPlayer` in *CardGame*, `CardGameInfoPresentation` for *CardGameGUI*.

Considering this example, it can be seen that beside the *0* category, three other categories can be identified that exist in most software systems (see Fig. 3a):

– *Application* (*A*): containing only application software, i.e., *CardGame*, *SheepsHead* and *CardGameGUI*.



+	DT	D	T	DG	0‘
DT	DT	DT	DT	DT	DT
D		D	DT	D	D
T			T	T	T
DG				DG	DG
0‘					0‘

Fig. 4. Addition of software categories.

- *Technical* (T): containing only technical software, e.g., Java *Swing* classes.[3]
- Combination of A and T (AT): e.g., *CardGameGUI-Swing* since it refines both an A (*CardGameGUI*) and a T (`Swing`) category.[4]

[14] summarizes the characteristics and rules for the software categories as follows: the categories are partially ordered, i.e., every category can refine one or more categories. The emerging category graph is acyclic. The category *0* (Zero) is the root category, containing global software. A category C is pure, if there is only one path from C to *0*. Otherwise, the category is impure. In Fig. 3a only the category AT is impure, since it refines the two categories A and T. All other categories are pure.

Terminology. We call a class that has the category C a C-class. Following from the category graph in Fig. 2 there are: AT-classes, A-classes, T-classes and *0*-classes. For the sake of readability, we do not explicitly mention interfaces, albeit what applies to classes applies to interfaces as well.

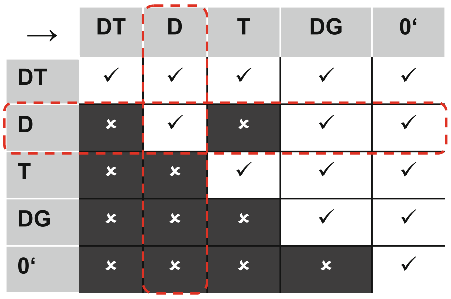
3 Categories for Generative Software

While [14] aims for finding components from the defined software categories, the goal of this paper is to determine whether a specific class should be *generated or not* by analyzing its dependencies to other classes.

To illustrate this, consider the following example. When having a class `Book` and a class `Jupiter`, which of these classes are generation candidates? Of course, it *depends on the domain*. If the domain is about planets, probably `Jupiter` is a candidate. In a carrier media domain, `Book` would be a candidate. So, we can say,

[3] Note that `Swing` classes are global (belonging to the JDK) and well-tested; hence meet the criteria of the category *0*. But –as usually the user-interface should be exchangeable– `Swing` classes are not necessarily global in a specific software system.

[4] In [14] also the *Representation* (R) category is presented. This category contains only software for transforming A category software to T and vice versa. It is a kind of cleaner version of AT. To demonstrate our approach, the R category can be neglected.



→	DT	D	T	DG	0‘
DT	✓	✓	✓	✓	✓
D	×	✓	×	✓	✓
T	×	×	✓	✓	✓
DG	×	×	×	✓	✓
0‘	×	×	×	×	✓

Fig. 5. Allowed dependencies between categories.

that a generation candidate somehow relates to the domain. But this condition is not enough. In a library domain where different books exist, `Book` would rather be general for the *whole domain* and should probably not be generated at all. Hence, additionally to the domain affiliation, a generation candidate *is not general for the whole domain*. Technically speaking, the class or interface should depend on a specific model (or model element). Consequently, a change in the model can imply the change of the generated class. Usually, classes that are global for the whole domain are not affected by changes in a model.

We adjusted the category model in Fig. 3a to better fit in with the domain. Figure 3b shows the modified category model.

The category A from Fig. 3a is renamed to D (Domain), to emphasize the domain. Consequently, the mixed form AT (Application and Technical) becomes DT (Domain and Technical). Category T remains unchanged. The new category DG (domain global) indicates software that is global for the whole domain and helps to differentiate from D-classes that are specific to the domain (a particular book, e.g., `CookBook`).

Resulting from the introduction of DG, the characteristic of the *0* category changes somewhat. It contains only global software that is well-tested and *independent of the domain*, e.g., `java.lang` and `java.util` of the JDK. To highlight the difference to the initial *0* definition, *0’* is used.

With the above objective in mind and upon searching for generation candidates, in particular, classes of the category D are interesting, i.e., D itself and DT, refining the category of both D and T (see Fig. 3b).

The matrix in Fig. 4 underscores which software category results if two categories are combined. A usage of *0’* has no effect, e.g., D + *0’* = D. The same is true for DG, as we defined it to be like *0’* (global for the whole domain). Hence, $D + DG = D$, e.g., if the D-class `CookBook` extends the DG-class `Book` it still remains a D-class. Only the combination of D and T leads to an (impure) mixed form, concretely DT. Any combination with DT results in DT, i.e., $* + DT = DT$.

The aim of this paper is not to define an architecture for generative software, but rather to use both the semantics of the software categories and the dependencies between the classes to find candidates for generated code. Nevertheless, this kind of categorization conforms to the concepts of the architecture reference model suggested by [8]. D, T and DT are what [8] call *Application*, consisting of generated and handwritten code. DG represents the *Business* and *Technical Platform*. Here, we do not distinguish between technical and domain platform, since this would not improve the final candidate list, but only complicate the category graph. *0'* corresponds to standard libraries of the programming language mentioned in [8]. However, the categories need not be as abstract as the architecture. They could be further divided if this improved finding generation candidates.

4 Dependency Rules for Categories

A total of four categories (plus the mixed form DT) have been suggested for a general classification of code in generative software (Fig. 3b). Classes of a particular category are only allowed to depend on classes of the same category and classes that are on the same path to *0'*. Consequently, based on these categories, the dependency matrix in Fig. 5 can be derived automatically.

The matrix can be read in two ways: line-by-line or column-by-column. The former shows the allowed dependencies *of* a category, whereas the latter shows the categories that may depend *on* a category. The first row in Fig. 5 shows that a DT-class may depend on classes of any of the categories. A D-class can only depend on D-, DG- and *0'*-classes[5] (Fig. 5, second row). A D-class must not depend on a DT-class. Only the other direction is allowed. Analogous to D-classes, a T- class may only depend on T-, DG- and *0'*-classes. A class from category DG cannot depend on any of the categories but DG and *0'*; otherwise it would contradict the definition of DG being global for the whole domain. For example, in the library domain, the (abstract) class `Book` (DG) would not know anything about the single books (such as `CookBook` (D) or `MDDBook` (D)). Of course, *0'*-classes can only communicate among each other. For instance, classes in the `java.lang` package (*0'*) do not have any dependencies to a class of any of the other categories.

As mentioned before, the columns in Fig. 5 show those categories that can depend on a specific category. It can be seen that this is somehow antisymmetric to the previously described allowed dependencies of a category.

In the following, we briefly describe the categorization in a formal way. $\mathbb{C}$ is a set containing the categories illustrated in Fig. 3b, i.e., $\mathbb{C} = \{DT, D, T, DG, 0'\}$. A pair (a, b) with $a, b \in \mathbb{C}$ means category *a directly* refines category *b*. Consequently, possible pairings are $\Re = \{(DT, DT), (DT, D), (DT, T), (D, D), (D, DG), (T, T), (T, DG), (DG, DG), (DG, 0'), (0', 0')\}$. For every category c, $\varphi(c)$ returns a set of categories which are directly refined by c. $\varphi(c)$ is defined as: $\varphi(c) = \{b | (c, b) \in \Re\}$ with $\varphi : \mathbb{C} \rightarrow P(\mathbb{C})$, where $P(\mathbb{C})$ is the power set of $\mathbb{C}$. $\varphi^*(c)$ is the corresponding *transitive closure*. The following listing shows the resulting set of $\varphi^*(c)$ for every category $c \in \mathbb{C}$:

[5] Note that a D-class that depends on a T-class is rather a DT-class.

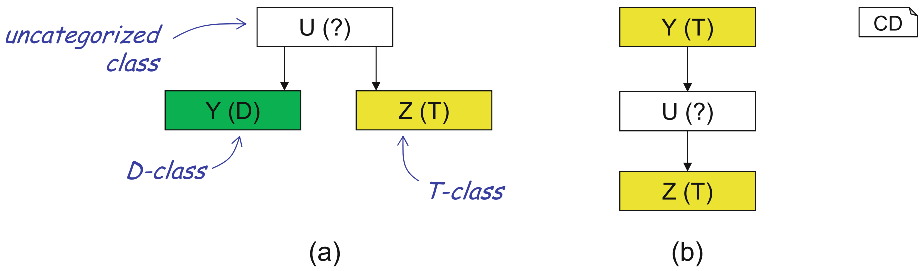


Fig. 6. (a) Uncategorized class depends on two categorized classes. (b) Uncategorized class in-between two categorized classes.

(1) $\varphi^*(DT) = \{DT, D, T, DG, 0'\}$
(2) $\varphi^*(D) = \{D, DG, 0'\}$
(3) $\varphi^*(T) = \{T, DG, 0'\}$
(4) $\varphi^*(DG) = \{DG, 0'\}$
(5) $\varphi^*(0') = \{0'\}$.

Analogously, $\varphi^{-1} : \mathbb{C} \rightarrow P(\mathbb{C}), \varphi^{-1}(c) = \{a|(a, c) \in \Re\}$ represents the set of categories that directly refine c, with $\psi^* = (\varphi^{-1})^*$ being its transitive closure:

(1) $\psi^*(DT) = \{DT\}$
(2) $\psi^*(D) = \{DT, D\}$
(3) $\psi^*(T) = \{DT, T\}$
(4) $\psi^*(DG) = \{DT, D, T, DG\}$
(5) $\psi^*(0') = \{DT, D, T, DG, 0'\}$.

With the help of these sets classes can be categorized. Figure 6a shows an example. The uncategorized class U depends on both the D-class Y and the T-class Z. Formally, the category of U is $\psi^*(D) \cap \psi^*(T) = \{DT, D\} \cap \{DT, T\} = \{DT\}$.

In Fig. 6b the uncategorized class U depends on a T-class. Additionally, a T-class depends on U. From this it follows that $\varphi^*(T) \cap \psi^*(T) = \{T, DG, 0'\} \cap \{DT, T\} = \{T\}$.

If two classes depend on each other, and thus, a circular dependency exists, they always have the same category. The reason is that $\varphi^*(c) \cap \psi^*(c) = \{c\}$ holds true for every category $c \in \mathbb{C}$, since the software category graph is acyclic.

In the above cases, the classes can be categorized unambiguously. Figure 7a demonstrates an example where the categorization is ambiguous. A DT-class depends on the uncategorized class which itself depends on a T-class: $\varphi^*(DT) \cap \psi^*(T) = \{DT, T\}$. Consequently, the category of U is either DT or T. In such a case, the class must be categorized manually. In general, the more classes are categorized, the better the categorization. For instance, if the class U additionally depends on a D-class (see Fig. 7b), its category will be determined unambiguously, since $\varphi^*(DT) \cap \psi^*(T) \cap \psi^*(D) = \{DT\}$.

Dependencies in Java. Up to now, we included the term dependency, but we did not define it so far. This is mainly because what a dependency ultimately

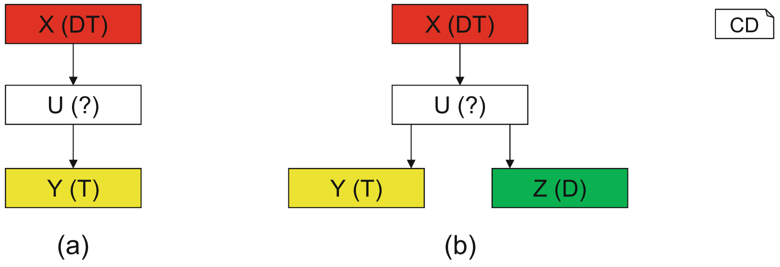


Fig. 7. (a) Ambiguous categorization. (b) Categorization becomes unambiguous by adding new classes.

is, depends on the (target) programming language. Java, for example, provides different kinds of dependencies between classes and interfaces. The following shows one possible classification, where the class `A` depends on the class `B` and the interface `I`, respectively:

- Inheritance: `class A extends B`
- Implementation: `A implements I`
- Import: `import B`
- Instantiation: `new B()`
- ExceptionThrowing: `throws B`
- Usage: field access (e.g., `b.fieldOfB`), method call (e.g., `b.methodOfB()`), declaration (e.g., `B b`), use as method parameter (e.g., `void meth(B b)`), etc.

These are dependencies in Java that are mostly manifested in keywords (e.g., `extends` and `throws`), and hence, hold for any Java software project. However, not all of these dependencies are always desired. It is important to determine first of all what a dependency ultimately is. For example, an unused import, i.e., a class that imports another class without using it, is not necessarily a dependency.

5 Categorization Approach

The suggested approach for the categorization of the source code is demonstrated in Fig. 8. Three inputs are needed for the categorization: the source code to be categorized (from which a dependency graph is derived), the category graph (such as in Fig. 3b) and an initial categorization of some of the classes and interfaces (usually done by hand). Using these inputs, a categorization tool analyzes the dependencies of the uncategorized classes and interfaces to the already categorized ones. With the information obtained from the category graph some of the uncategorized classes and interfaces can be categorized automatically. For example, if a class `C` depends on a D- and a T-class and the category graph in Fig. 3b is given, the category of class `C` is definitively DT, since only this category refines both D and T.

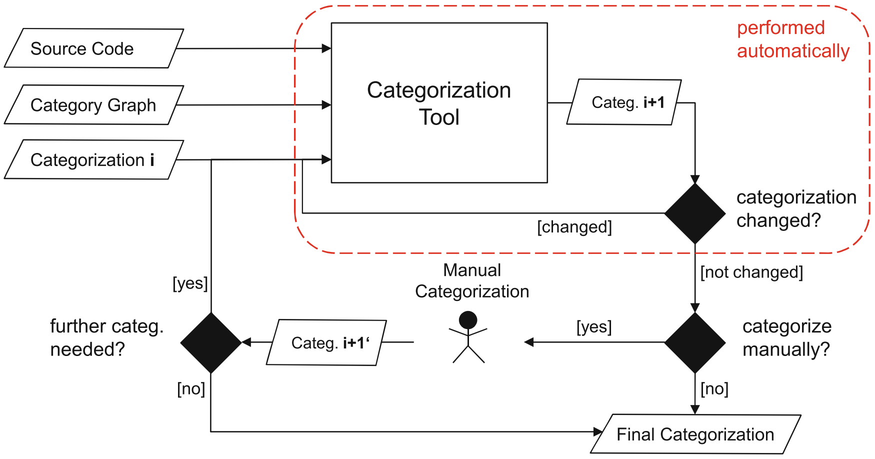


Fig. 8. Overview of the categorization approach.

In some cases the order of the categorization process matters. For example, if a class `A` only depends on a class `B` (and no categorized class depends on `A`), `A` will not be categorized until `B` is categorized. To prevent that the order has an effect on the final categorization, the categorization is performed iteratively. The output of iteration i serves as input for the next iteration $i{+}1$. This is repeated until a fixpoint is reached, that means, no further classes and interfaces could be categorized. These iteration steps can be conducted fully automatically. If there are still uncategorized classes left, some of them can be categorized by hand (Sect. 6 illustrates this case by an example). This updated categorization, again can serve as input. The process can be repeated until the whole source code is categorized or no further categorization is needed. Finally, classes and interfaces with a specific categorization serve as candidates for code to be generated. Here, this applies to the categories D and DT. The user now can decide which of these candidates will become generated code.

Concept Realization. The categorization tool consists of two domain-specific languages developed with the MontiCore language workbench [15–17]. The first DSL allows to specify a software category graph. The restrictions defined in Sect. 2 (e.g., acyclic graph) are checked by intra-model context conditions [6]. With the second DSL the source code can be categorized by mapping the defined categories to Java classes. The DSLs are composed through mechanisms described in [18–20].

As described in Sect. 6, the categories are iteratively extended. Technically, this means, the mapping model is extended after each iteration. Using an explicit DSL to categorize the classes has several advantages. First, when the developer initially categorizes the classes by hand, static checks can be performed to support the user, for example, that a class cannot be categorized with more than one category. Second, the developer can use the same tools and user-friendly syntax for the initial categorization of the classes as well as the manual categorization after each iteration.

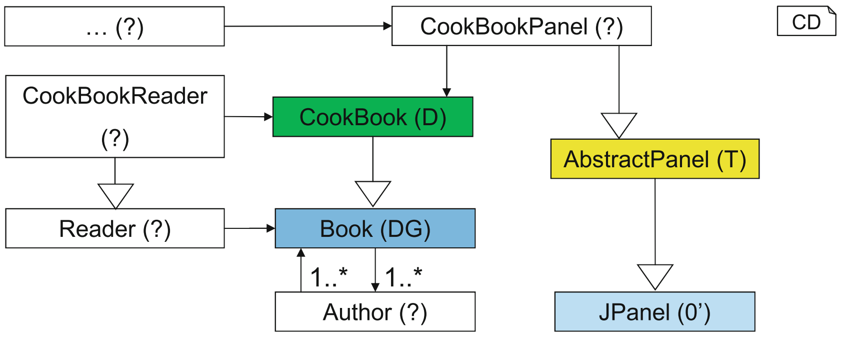


Fig. 9. Initially categorized classes.

6 Analyzing Usefulness on an Illustrative Example

Now, with the help of the allowed dependencies defined in Sect. 4, given some classes, the category of each of the classes can be derived semi-automatically, following the approach presented in the previous section.

Consider the case in Fig. 9. The figure depicts overall ten classes, whereby four are pre-categorized (`CookBook`, `AbstractPanel`, `Book` and `JPanel`) and six are not. The category is in parentheses beside the class name. Uncategorized classes are marked with a question mark `(?)`. Let us assume that the four categorized classes already exist and are categorized (e.g., manually by an expert) and the six other classes are newly created. This situation can arise, for instance, when software evolves. In the following, the categorization process is illustrated.

The class `CookBookPanel` communicates with both a D-class (`CookBook`) and a T-class (`AbstractPanel`). Following Fig. 5, only a DT-class may communicate with a D as well as with a T class (marked by a check mark in the D and T column). Thus, `CookBookPanel` is definitively a DT-class. Moreover, any other class depending on `CookBookPanel` (represented by the three dots), is also a DT-class. In the column DT in Fig. 5 there is only a check mark for DT.

Next, `CookBookReader` depends on the D-class `CookBook` and the not yet categorized class `Reader`. If `Reader` is a DT- or T-class, `CookBookReader` will be definitive a DT-class, for it would depend on a D-class and either a DT- or T-class. With regard to Fig. 5, this only fits for DT-classes. If `Reader` is of any of the other categories, `CookBookReader` will be a D-class. However, when trying to categorize `Reader`, we encounter a problem. `Reader` only depends on `Book`, a DG-class. According to Fig. 5 this can apply to any category except *0'*. So, in this iteration, `Reader` cannot be categorized automatically. Consequently, the exact categorization of `CookBookReader` cannot be determined.

Analogous to the class `Reader`, the class `Author` only depends on the DG-class `Book`. So, except *0'*, it can be of any category. Unlike the previous case, `Book` also has a dependency to `Author`, which means that `Author` is either DG or *0'*. We have already excluded *0'*; hence, only DG remains as a possible category for `Author`. Figure 10 shows the extended categorization after this iteration.

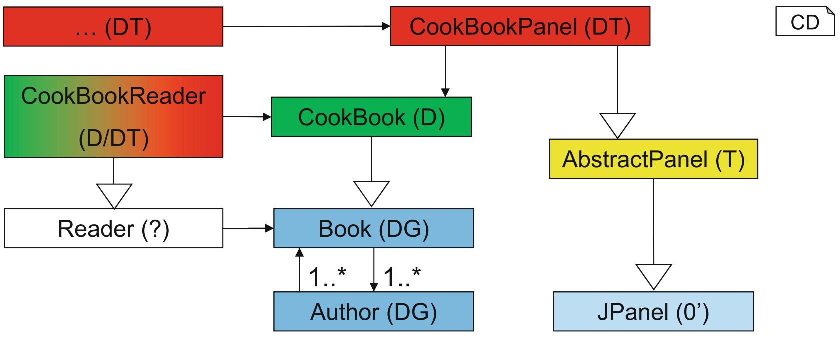


Fig. 10. Categorized classes after the first iteration.

Two classes could not be categorized exactly after the first iteration: `Reader` and `CookBookReader`. Recalling that our goal is to find generation candidates, we are above all interested in classes of the category D. So, the approximate categorization of `CookBookReader` (D or DT) is sufficient, since both D and DT are of the category D. In contrast, `Reader` is still completely uncategorized which hampers the categorization of classes depending on it. There are two options to categorize `Reader` in the next iteration: either manually by the expert or automatically by adding new classes and dependencies limiting the possible categories of `Reader`.

Note that the order of the categorization of `CookBookPanel` and the classes depending on it (marked by "…") is important for the first iteration. The "…" classes could not be categorized if they were considered *before* `CookBookPanel`. However, the order has no impact on the final result, since after the first iteration `CookBookPanel` is surely categorized, and thus, the "…" classes can be categorized in the next iteration.

Finally, three candidates (plus the "…" classes) for generated code are identified: `CookBook` (D), `CookBookReader` (D/DT) and `CookBookPanel` (DT). All of these classes belong to the category D directly or indirectly (i.e., DT), and hence, are somehow related to the domain. Having these candidates, the generator developer has to decide which of these classes in the end need to be generated and which remain handwritten. Of course, this decision is restricted above all by the information content of the input model. The generator developer must be aware of this restriction.

Please note that without knowing anything about the intrinsic properties of the uncategorized classes, the classes can be categorized by only analyzing the dependency graph. Of course, this is not always possible. Yet the more classes are already categorized, the better the proposed categorization of the uncategorized classes will be.

Detecting Wrong Categorization. A wrong initial categorization, e.g., if `CookBook` was categorized as *0'*, can lead or at least hamper the final categorization. One reason for wrong categorization is the evolution of software where classes can change and with them their categories. Another source of error is the manual categorization done initially or after each iteration. To tackle this issue, consistency checks as described, for instance, in [21–23] are suitable. From the dependency matrix in Fig. 5 rules can be derived to check the consistency of the source code and the category graph.

7 Further Dependencies

Up to now, only the technical dependencies of the code are considered for finding generation candidate classes (see Sect. 4). So, for example, if a class `A` extends a class `B`, it depends on that class `B`. But dependencies are not necessarily manifested technically. There can be further dependencies, such as *naming dependencies*. Figure 11 shows an example. The class `CookBookWindow` does not have any technical dependencies on `CookBook`, that means, it neither extends `CookBook` nor does it have any associations to it. It extends `AbstractWindow`, a T-class, hence, it is probably a T-class, too. However, `CookBookWindow` contains the name of `CookBook` as prefix in its class name. Of course, this can be purely coincidental. But, if there are specific conventions dictating that the name of a window for a class is composed of the name of that class and the suffix "Window", the naming dependency in Fig. 11 is very likely no coincidence. Considering this naming dependency, `CookBookWindow` has both a dependency to the T-class `AbstractWindow` and the D-class `CookBook`. Consequently, `CookBookWindow` is a DT-class and a generation candidate. Please note that from the architecture's point of view a (technical) dependency between `CookBookWindow` and `CookBook` might be forbidden. Hence, deriving the dependency rules from the architecture (and not from software category graph) would limit the kinds of possible dependencies.

In Fig. 11 there is a further clue that `CookBookWindow` refers to the `CookBook` class: its constructor has the same types as the attributes in `CookBook`. Hence, the dependencies between classes can be atomic (such as extension) or more complex, matching a specific pattern (for example similarity [24] and design patterns [25]). In sum, what a dependency finally is, depends on the software system and its conventions. This affects the emerging dependency graph of the source code and can also lead to a different candidate list. In any case, the procedure as described in Sects. 5 and 6 remains unchanged.

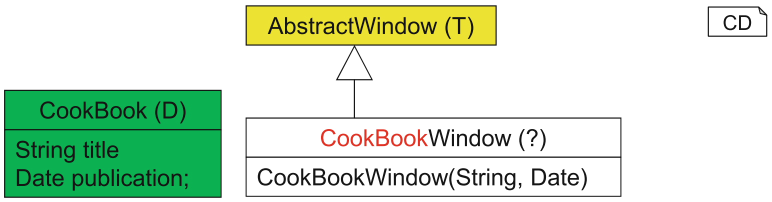


Fig. 11. Naming dependency between classes.

8 Related Work

[26] introduce class-level micro-patterns for common Java programming practices, such as immutability. Each pattern is assigned to a main category and optionally to an additional category. The categories in the approach are similar to our software categories such as they (logically) group together classes based on specific patterns and dependencies, respectively. *TaxTOOL* [27] and *TaxTOOLJ* [28] use a comprehensive taxonomy of OO characteristics (e.g., polymorphism) to classify classes. [29] present an approach to find so called *key classes* of a software project, based on dynamic coupling and webmining. Key classes are identified by their dependencies to other classes, since they "are tightly coupled with other parts of the system".

In [30–32] an approach is presented for finding stereotypes of methods and classes of a software system. Method stereotypes describe characteristics of a method, such as accessor (e.g., getter), mutator (e.g., setter), and creational (e.g., factory). A class stereotype categorizes a class, e.g., `DataClass`. The stereotype of a class depends on the stereotype of its methods. Rules exist for each kind of stereotype. Based on these rules, the concrete stereotypes of methods and classes are determined. Our approach has a lot in common with the one presented in [30–32]. Both approaches conduct static code analysis in order to categorize code (elements). Predefined rules serve as basis. However, the main difference is that in our approach dependencies between (un-)categorized classes are analyzed in order to determine the categorization. Consequently, a higher amount of categorized classes can improve the result. In contrast, method stereotypes, for example, rely on a method's characteristics independent of how it relates to other methods.

Some visualization approaches are presented by [33,34] which enrich classes and methods, respectively, with semantic information in order to improve code understanding.

All approaches have in common that they categorize or classify code using specific rules or patterns. These rules and patterns can be combined with our approach in order to categorize classes.

9 Conclusion

Code generators are crucial to MDD, transforming abstract models to executable source code. The generated source code often depends on handwritten code, e.g., code from the domain framework. When a code generator is developed or evolved, the generator developer manually decides which classes need to be generated and which remain handwritten. This task can be time-consuming, labor-intensive and may generate more code than is necessary, hampering the maintenance of the software.

This paper has introduced an approach that can aid the generator developer in finding candidates for generated code. First, a software category graph is defined. From this graph the allowed dependencies between the corresponding classes (and interfaces) are derived automatically. After an initial categorization of some classes, further classes can be categorized automatically, by analyzing their dependencies. This procedure is conducted iteratively until all classes are categorized or no more categorization is needed. Finally, generation candidates are all classes belonging to the domain categories.

References

1. Mir Seyed Nazari, P., Rumpe, B.: Using software categories for the development of generative software. In: Hammoudi, S., Pires, L.F., Desfray, P., Filipe, J.F. (eds.) Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, pp. 498–503. SciTePress, Angers (2015)
2. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: the Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc, Boston (2003)
3. Rumpe, B.: Modellierung mit UML, 2te edn. Springer, Berlin (2011)
4. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Future of Software Engineering 2007 at ICSE, pp. 37–54 (2007)
5. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley, Hoboken (2008)
6. Schindler, M.: Eine werkzeuginfrastruktur zur agilen entwicklung mit der UML/P. In: Aachener Informatik Berichte, Software Engineering. Shaker Verlag (2012)
7. Krahn, H., Rumpe, B., Völkel, S.: Roles in software development using domain specific modelling languages. In: Gray, J., Tolvanen, J.P., Sprinkle, J., (eds.) Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling 2006 (DSM 2006), Volume TR-37 of Technical report., Portland, Oregon, USA, pp. 150–158. Jyväskylä University, Finland (2006)
8. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. Wiley, Chichester (2006)
9. Fowler, M.: Domain Specific Languages. Addison-Wesley Professional, Boston (2010)
10. Rumpe, B.: Agile Modellierung mit UML. Xpert Press, 2nd edn. Springer, Berlin (2012)
11. Greifenberg, T., Hölldobler, K., Kolassa, C., Look, M., Mir Seyed Nazari, P., Müller, K., Navarro Perez, A., Plotnikov, D., Reiss, D., Roth, A., Rumpe, B., Schindler, M., Wortmann, A.: A comparison of mechanisms for integrating handwritten and generated code for object-oriented programming languages. In: Hammoudi, S., Pires, L.F., Desfray, P., Filipe, J.F. (eds.) Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, pp. 74–85. SciTePress, Angers (2015)
12. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading (2002)
13. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus (2005)
14. Siedersleben, J.: Moderne Software-Architektur: Umsichtig Planen, Robust Bauen mit Quasar. Dpunkt.Verlag GmbH, Heidelberg (2004)

15. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: MontiCore: a framework for the development of textual domain specific languages. In: 30th International Conference on Software Engineering (ICSE 2008), Companion Volume, Leipzig, Germany, 10–18 May 2008 (2008)
16. Krahn, H., Rumpe, B., Völkel, S.: Monticore: modular development of textual domain specific languages. In: Paige, R., Meyer, B. (eds.) TOOLS EUROPE 2008. LNBIP 11, vol. 11, pp. 297–315. Springer, Heidelberg (2008)
17. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. Int. J. Softw. Tools Technol. Transf. (STTT) **12**, 353–372 (2010)
18. Völkel, S.: Kompositionale entwicklung domänenspezifischer sprachen. In: Number 9 in Aachener Informatik-Berichte, Software Engineering. Shaker Verlag (2011)
19. Look, M., Navarro Pérez, A., Ringert, J.O., Rumpe, B., Wortmann, A.: Black-box integration of heterogeneous modeling languages for cyber-physical systems. In: Combemale, B., De Antoni, J., France, R.B. (eds.) Proceedings of the 1st Workshop on the Globalization of Modeling Languages (GEMOC). CEUR Workshop Proceedings, vol. 1102, Miami, Florida, USA (2013)
20. Haber, A., Look, M., Mir Seyed Nazari, P., Navarro Perez, A., Rumpe, B., Völkel, S., Wortmann, A.: Integration of heterogeneous modeling languages via extensible and composable language components. In: Hammoudi, S., Pires, L.F., Desfray, P., Filipe, J.F. (eds.) Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, pp. 19–31. SciTePress, Angers (2015)
21. Murphy, G.C., Notkin, D., Sullivan, K.J.: Software reflexion models: bridging the gap between design and implementation. IEEE Trans. Softw. Eng. **27**, 364–380 (2001)
22. Terra, R., Valente, M.T.: A dependency constraint language to manage object-oriented software architectures. Softw. Pract. Experience **39**, 1073–1094 (2009)
23. Passos, L., Terra, R., Valente, M., Diniz, R., Mendon, N.: Static architecture-conformance checking: an illustrative overview. IEEE Softw. **27**, 82–89 (2010)
24. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. IEEE Trans. Softw. Eng. **33**, 577–591 (2007)
25. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Reading (1995)
26. Gil, J.Y., Maman, I.: Micro patterns in java code. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. OOPSLA 2005, pp. 97–116. ACM, New York (2005)
27. Clarke, P., Malloy, B., Gibson, P.: Using a taxonomy tool to identify changes in OO Software. In: Proceedings of Seventh European Conference on Software Maintenance and Reengineering, 2003, pp. 213–222 (2003)
28. Clarke, P.J., Babich, D., King, T.M., Kibria, B.G.: Analyzing clusters of class characteristics in {OO} applications. J. Syst. Soft. **81**, 2269–2286 (2008)
29. Zaidman, A., Demeyer, S.: Automatic identification of key classes in a software system using webmining techniques. J. Softw. Maintenance Evol. Res. Pract. **20**, 387–417 (2008)
30. Dragan, N., Collard, M.L., Maletic, J.I.: Reverse engineering method stereotypes. In: Proceedings of the 22nd IEEE International Conference on Software Maintenance. ICSM 2006, pp. 24–34. IEEE Computer Society, Washington (2006)
31. Dragan, N., Collard, M.L., Maletic, J.I.: Using method stereotype distribution as a signature descriptor for software systems. In: 2009 IEEE International Conference on Software Maintenance, ICSM 2009, pp. 567–570 (2009)

32. Dragan, N., Collard, M.L., Maletic, J.I.: Automatic identification of class stereotypes. In: Proceedings of the 2010 IEEE International Conference on Software Maintenance, ICSM 2010, pp. 1–10. IEEE Computer Society, Washington (2010)
33. Lanza, M., Ducasse, S.: A categorization of classes based on the visualization of their internal structure: the class blueprint. In: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2001, pp. 300–311. ACM, New York (2001)
34. Robbes, R., Ducasse, S., Lanza, M.: Microprints: a pixelbased semantically rich visualization of methods. In: In Proceedings of ESUG 2005, 13th International Smalltalk Conference, pp. 131–157 (2005)