Function References as First Class Citizens in UML Class Modeling

Steffen Heinzl and Vitaliy Schreibmann
University of Applied Sciences Würzburg-Schweinfurt, Sanderheinrichsleitenweg 20, 97074 Würzburg, Germany

Keywords: Functional Modeling, UML, MOF, Modeling.

Abstract: There have been a number of philosophical discussions whether functional programming or object-oriented programming is the better programming concept. In reality, programmers utilize both concepts and functional programming concepts improve object-oriented languages. Likewise the modeling of OO languages should also reflect these concepts in the modeling process. This paper improves the modeling of behavior (usually expressed through functional programming) in UML class diagrams. In UML class diagrams, behavior modeling is only achieved by modeling a new class containing the desired function. If several alternatives for a certain behavior have to be expressed, the modeling complexity increases because we need to introduce an interface and for each alternative an additional class. Therefore, we propose a new *function* element that circumvents these problems and reduces the complexity of the model. Due to the proposed `<<Function>>` stereotype, functions in the model can be identified at first glance. The new model is motivated by the strategy pattern and evaluated against a more complex design pattern. A possible first implementation is presented.

1 INTRODUCTION

Object-oriented programming (OOP) is used to map objects as well as relations between objects from the real-world into programming languages. During this mapping process, developers have repeatedly faced a number of similar problems. Gamma et al. (Gamma et al., 1995) collected object-oriented design patterns that had been used to overcome these recurring problems. Developers following these design patterns possess a common vocabulary when talking about these problems as well as a common way to structure a solution. Some of the patterns (such as the strategy pattern) are cumbersome due to the lack of expressiveness of purely object-oriented languages and their representation by class diagrams.

For a few years now, all major OOP languages (Java, C#, C++) (partially) support functional programming and allowing developers to mix both programming styles. But are purely object-oriented design patterns still adequate for modeling? Should modeling languages, such as the Unified Modeling Language (UML) (Object Modeling Group, 2015), contain functional concepts?

The contributions of this paper are answers to these questions by introducing a new UML stereotype `<<Function>>` to model function references as first-class citizens in UML class diagrams. The proposed *function* element complements the modelers' existing toolkit (consisting of interfaces, classes, enums, etc.). The evaluation shows that the complexity of modeling design patterns can also be reduced.

The paper is structured as follows: Section 2 explains—exemplified by the strategy pattern—the reason for introducing a *function element*. In the Section 3, the *function* element is specified. Section 4 shows how the function element can reduce modeling complexity. Section 5 demonstrates a first implementation of the model in Java and Section 6 presents the related work. Section 7 concludes our paper and discusses areas of future work.

2 MOTIVATION

Dijkstra (Dijkstra, 1984) claimed: "the Romans have taught us "Simplex Veri Sigillum" that is: simplicity is the hallmark of truth [...], but [...] the sore truth is that complexity sells better". The strategy pattern shown in Figure 1 is a good example for overly complex depiction compared to its function.

The strategy pattern consists of a Context that has an object of a strategy. The Strategy is represented by an abstract class or interface, and a class hierarchy subclassing/implementing the abstract class/interface. ConcreteStrategyA to ConcreteStrategyC encapsulate different behaviors by providing a single accessible

Heinzl, S. and Schreibmann, V.
Function References as First Class Citizens in UML Class Modeling.
DOI: 10.5220/0006783603350342
In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, pages 335-342
ISBN: 978-989-758-300-1
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

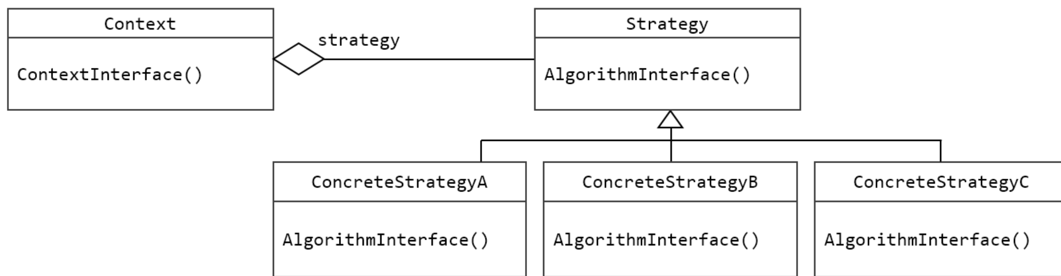

Figure 1: Strategy Pattern.

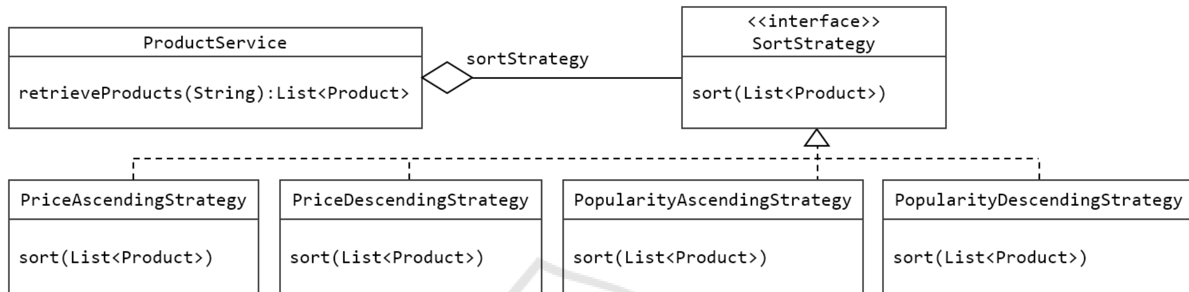

Figure 2: An example of the strategy pattern taken from an online shop.

method. Most of the times, the pattern is used for the choice of an algorithm or—in more general—to capture different alternative behaviors. According to Gamma et al. (Gamma et al., 1995, pp. 315) the pattern is used to prevent the use of multiple conditionals to select an algorithm. Instead the algorithms can be varied independently of the client. New algorithms can be added by adding a new subclass/implementing class. Modeling the strategies in such a way is fine if you want to put strong emphasis on the different strategies. For example, Figure 2 depicts a sorting algorithm for an online shop that is able to sort products in an ascending or descending fashion by means of the products' prices or popularity.

When modeling a complete online shop, concepts such as customer, invoice, cart, order, message formats, services, etc. might be more important than sorting strategies, which might take a large space in the shop model.

The strategy pattern has been selected as an example because each strategy consists of a straightforward class encapsulating only behavior without state. Behavior can usually also be encapsulated in a single method in a separate encapsulating object. Most languages provide function references that also can encapsulate an anonymous method, a so-called lambda expression. In Java, for example, function references are introduced by a FunctionalInterface named `Function<T,R>`. `T` stands for the type of the argument passed to the function and `R` stands for the return type of the function. C# provides a similar solution. Function references in C# are introduced by a so-called Delegate named `Func<T,TResult>`. Similarly, `T` is the type of the incoming argument, `TResult` the type of the outgoing result.

UML class diagrams are limited to only classes. Stereotypes are necessary to distinguish between classes, abstract classes, interfaces, etc. Anonymous methods, functional interfaces, functional references, and delegates are all terms relating to the realization of lambda expressions in OOP languages. But UML class diagrams do not support this concept out-of-the-box. Therefore, we propose to define a stereotype `<<Function>>` to model behavior instead of using the `<<interface>>` stereotype (and subclassing as in the strategy pattern). Beside classes, interfaces, etc. we now have a new model element named *function* in our toolkit. With the `<<Function>>` stereotype shown in Figure 3 we can model the types and return types the function reference is able to work with. Furthermore, the element in Figure 3 contains a function reference attribute to a *function* (i.e. the UML class with the stereotype `<<Function>>`).

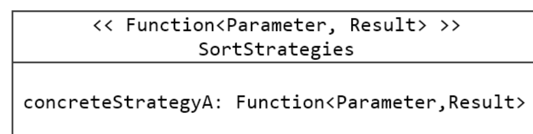

Figure 3: Function stereotype with one function reference.

The function element acts a container to save a function reference that is able to work with the

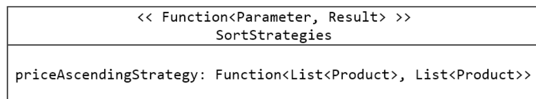

Figure 4: Function stereotype example with one function reference.

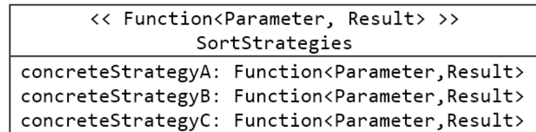

Figure 5: Function stereotype with several function references.

type `Parameter` and return type `Result` given in the stereotype. Figure 4 shows an example *function* from our online shop which contains a function reference taking a list of products and returning a sorted products list.

We can also group function references as shown in Figure 5. The stereotype specifies that all function references work with the parameter and result type specified by the stereotype. That means the stereotype forces all Function objects to be of type `Function<Parameter,Result>`.

After all this effort, we can now take a look again at the strategy pattern in Figure 6, this time expressed with a *function*. The context has an aggregation to SortStrategies. This aggregation means—depending on the multiplicity—that the Context can have between 0 and n function references with the type defined in the stereotype. Usually this attribute is filled with the Functions attributes listed inside the *function* by holding a direct reference (i.e. concreteStrategyA).

3 SPECIFICATION OF THE FUNCTION ELEMENT

In UML, we apply stereotypes to "transform" classes to either interfaces, abstract classes, enums, or others. Our approach utilizes this mechanism to introduce the *function* element. Until now, it has not been possible to directly associate function references with other elements in class diagrams. With the *function*, it is possible to treat function references as first class citizens in class diagrams.

3.1 Function Element

In the following, we specify the *function* and its semantics.

In Figure 7, a general representation of the *function* is shown. The *function* has a name (*FunctionName*), a stereotype with parameter and result type, and several attributes with a type specified by the stereotype. The following sections discuss the components of *function* in detail:

Name: The name serves—as usual—as a good description for the element in order to provide a clear understanding of the reality captured in the model.

Stereotype: Our stereotype is named Function and enables the modeler to distinguish between *function* and normal classes. Furthermore, the Function stereotype can be parameterized to specify the object types that **all** its function references take as parameter and return types. In case the stereotype is not parameterized, the attributes (i.e. function references) must indicate the parameter and the return type themselves.

Attributes: The attributes of a *function* are static in nature because they are known at coding time. As a consequence, the attributes are all non-modifiable and therefore read-only. The attributes are by default public and the readOnly property can be omitted. Also, the type of the attribute can be omitted if it is already specified by the stereotype. Such a specification is depicted in Figure 8. Non function references are not allowed. In case a state is needed across several function references, a normal class should be used for modeling.

The parameters have a polymorphic character. That means that either the type of the parameter itself or of a subclass can be used.

If several arguments are to be passed to the function reference, they should be encapsulated in a single object. Alternatively, it would also be possible to use a parameter list and interpret the last parameter as the return type. As a remark: Common programming languages such as C# and Java have decided against this approach and work instead with a single parameter type and a single return type.

3.2 Multiplicity and Relations

An element can be related to a number of functions in the same ways it can be related to other objects. In Figure 9, an instance of Context holds exactly *n* function references specified in our *function*. The main difference to normal classes is that not *n* objects of *FunctionName* are held by the Context instance but n **attributes** (i.e. function references) inside of *FunctionName*. All known multiplicities from UML can be utilized with the multiplicity of *n* representing several function references.

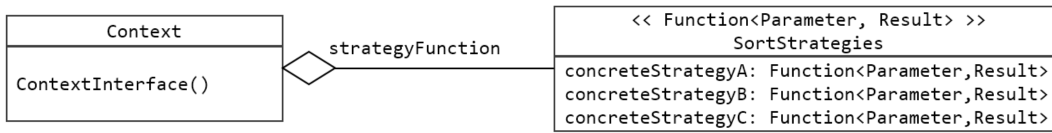

Figure 6: Strategy pattern expressed with the new *Function*.

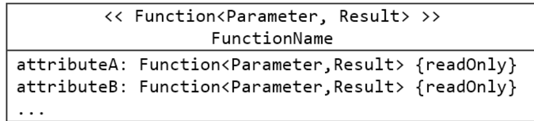

Figure 7: Specification of the function.

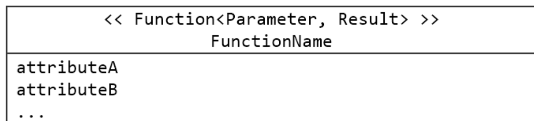

Figure 8: Specification of the Function without Explicitly Showing Defaults.

3.3 Abstraction

The Function<Parameter,Result> (or shorter: Function<P,R>) provides a good abstraction for different languages. Therefore, the knowledge of language specific types is not necessary in the model.

We provide two examples how the abstraction matches types from the underlying languages: The following mapping can be used to map our abstraction to Java types from the standard library:

- Function<P, void> matches Consumer<P>
- Function<void, R> matches Supplier<R>
- Function<P,R> matches Function<P,R>
- Function<void,void> matches Runnable

The following mapping can be used for C#:

- Function<P, void> matches Action<P>
- Function<void, R> matches Func<R>
- Function<P,R> matches Func<P,R>
- Function<void,void> matches Action

3.4 MOF Extension

Our proposed approach can be realized by an extension of the EMOF model from the Meta Object Facility (MOF) Specification (Object Management Group (OMG), 2016, p. 27). We added the new *function* element on the same level as the class element using a similar extension mechanism that has already been shown by Min et al. (Min et al., 2011). The reason is that a *function* element in the UML model does not share all properties and behavioral aspects of a class, i.e. for example the list of operations. In addition, we can extend the *function* element independently from the class element (e.g. for the definition of stereotypes and subelements). We decided to keep the extension of MOF (as shown in Figure 10) to a minimum and extend it further in future work.

4 EVALUATION

As an evaluation for our approach, we take a look at three different examples and verify the application of the newly introduced *function*.

4.1 Visitor Pattern

We selected the rather complex visitor pattern from (Gamma et al., 1995, pp. 334) (depicted in Figure 11) to analyse how well our modeling approach is suited to simplify the modeling.

We start modeling by adding different function references to a *function* element. The small excerpt in Figure 12 already shows a number of problems:

The Element and its subclasses accept a visitor with the accept method. The problem is that the Element in our modeling has to accept function references of different parameters and result types (see Figure 12). We have to use a type, such as Function<Element, void> instead of Function<ConcreteElementA, void> or Function<ConcreteElementB, void>. In a programm, this distinction is usually done during runtime using polymorphism and late binding. The problem lies in the access to generic types during runtime, which is not provided by all widespread object-oriented languages. For example in Java, type erasure is applied to generic types during compilation. This point must be embraced in an implementation of the model as discussed in Section 5 and shown in Figure 13.

The accept method in the Element interface and the ConcreteElement classes are replaced by the standard method for executing a Function, and hence need not be modeled explicitly. The two visitor function references offered can work on all subclasses of Element. The complexity of the model is reduced compared to (Gamma et al., 1995, pp. 334).

The visitor pattern is usually used to visit an object structure. Between the visits of single elements of the

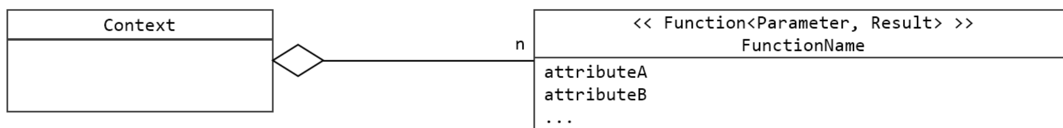

Figure 9: Example of multiplicity with function element.

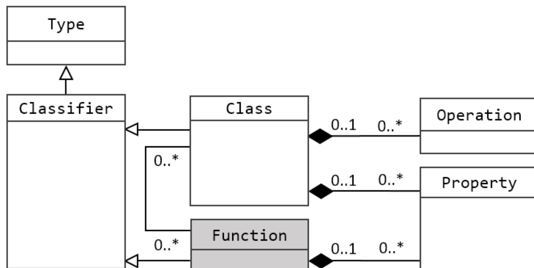

Figure 10: Simplified depiction of the MOF extension with a function element.

structure, the visitor is also able to save state. Saving state is problematic in the functional approach but can be achieved by passing the state as an argument from function invocation to function invocation.

4.2 "Simple" Visitor

The proposed *function* element can also be used for simpler and more often used behavioral modeling. Figure 14 shows one way to instantiate a thread in Java. The constructor of the `Thread` class accepts a "visitor" that implements the interface `Runnable` and executes the `Runnable`'s `run` method in its own `run` method. Figure 15 illustrate the same situation but modeled with a *function*. A Java Runnable has exactly one method (the `run` method), which takes and returns no arguments perfectly matching `Function<void, void>`. This model is elementary and appears more often in software compared to the full visitor pattern. Since `Function<void, void>` can automatically mapped to `Runnable`, we do not—in contrast to the first model—explicitly need to show the interface in the model.

4.3 Strategy Pattern

To end our evaluation, let us point to the Motivation section where the less complex model of the strategy pattern (see Figure 6) was contrasted to the original model. This example has shown very well how modeling was made less complex.

5 IMPLEMENTATION

In the Evaluation section, we spotted the problem that in Java generic types are erased by the compiler and are not available at runtime. We could achieve the desired behavior by broadening the interface through changing the `accept` method's signature to `accept(Function<ConcreteElementA, void>, Function<ConcreteElementB, void>)`. The problem with this approach, however, is that each time a new ConcreteElement is defined, we have to change the interface by adding another parameter to the accept method.

Mario Fusco has shown an implementation of the visitor pattern using functional programming in Java (Fusco, 2016d). His work inspired our implementation in Java that can be used to implement the suggested *function* element. For each *function* element in the model, we create a class with function references. Figure 18 shows an example for two different sort strategies. As a function reference, we use our class `UMLFunc` as shown in Figure 19 that implements the function interface.

The `UMLFunc` circumvents the problem that Java cannot access the generic type at runtime, allowing the registration of class-behavior pairs. Allowed classes are mapped to a behavior that should occur when the `UMLFunc` is executed on an object of that class. The standard execution method of a function in Java is the `apply` method. This method must be overriden to apply different functions based on the class name by looking into the map. We see one weakness of this implementation: We do not support true polymorphism. We register the implementation class of our list instead of the List or Collection interface and cannot distinguish between lists of different types (e.g. list of prices, list of customers).

For the visitor pattern, we need a `UMLFunc` for each different visitor. With the `register` method we specify a class (ConcreteElement1, ConcreteElement2, ...) with the corresponding behavior (e.g. a lambda expression calculation the area of a mathematical figure). So instead of carrying the name of each class in the method's name (e.g. visitConcreteElementA and visitConcreteElementB), the class is a parameter to the register method.

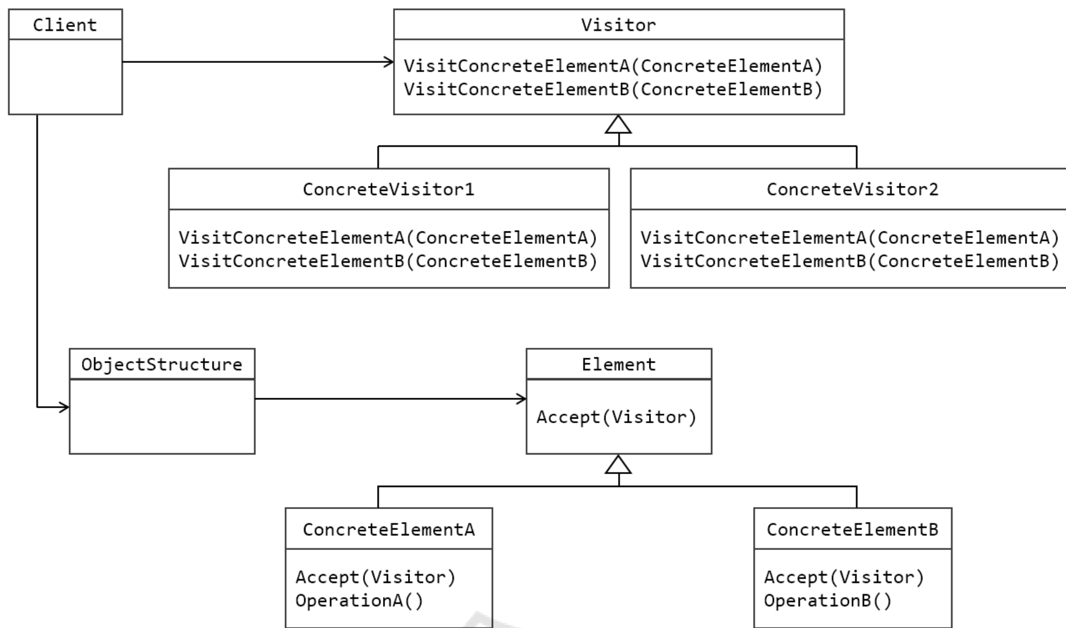

Figure 11: Depiction of the visitor pattern from the book (Gamma et al., 1995).

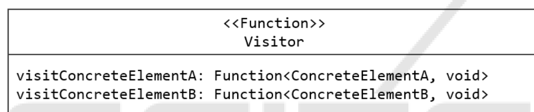

Figure 12: Excerpt of visitor pattern with function.

6 RELATED WORK

The aim of our work was to provide a modeling of function references for OOP languages which have been extended by functional programming concepts.

There are a few people, such as Mario Fusco (Fusco, 2016a) (Fusco, 2016b) (Fusco, 2016c) (Fusco, 2016d), who dealt with the topic of combining object-oriented programming design patterns with functional programming concepts that were brought into the OOP languages excessively. This shows nicely how complex examples of OOP work together with the functional programming concepts. He did, however, only address improvements when it comes to writing code but not for the modeling.

UML (Object Modeling Group, 2015) has a lot of capabilites. If we had to model the *function* element with current UML (version 2.5), we could use the abstract class/interface approach as shown for the strategy example in Figure 2. But the goal was to provide a more prominent modeling of function references, making them first class citizens in modeling. So we try to use normal UML model the newly introduced semantics as similarly as possible. First, we take a normal class with attributes. For every function reference, you have to specify the types it adheres to. The attributes have to be `{readOnly}` and *public* as shown in Figure 16, otherwise they could be overwritten during the course of a program execution. From a modeling perspective we can avoid several drawback with our approach:

1. With default UML parameter and result of the function references are not seen at first sight. However, including the parameter and result types in the stereotype enables a more prominently modeling.
2. With default UML it is harder to understand that the actual intention of the modeling is the encapsulation of behavior.
3. Usually, in order to model functional references, you have to take the actual type of the implementation. The function used in the stereotype abstracts from that. In Java, the name of the container for function references is also Function, but neither parameter nor result type could be void. Other types are needed to achieve this behavior and in C#, you would need to use Func. If *function* was a first class element in UML as we propse, tool support should easily achieve the mapping to different programming languages.

Our *function* element in UML simplifies the modeling of dynamic systems, such as mobile agents. Bosse (Bosse, 2016) defined a system of mobile agents in an Internet of Things environment with JavaScript as a programming language. An agent was modeled with an Activity-Transition Graph (ATG) (Bosse, 2016, Figure 2) which consist of a class

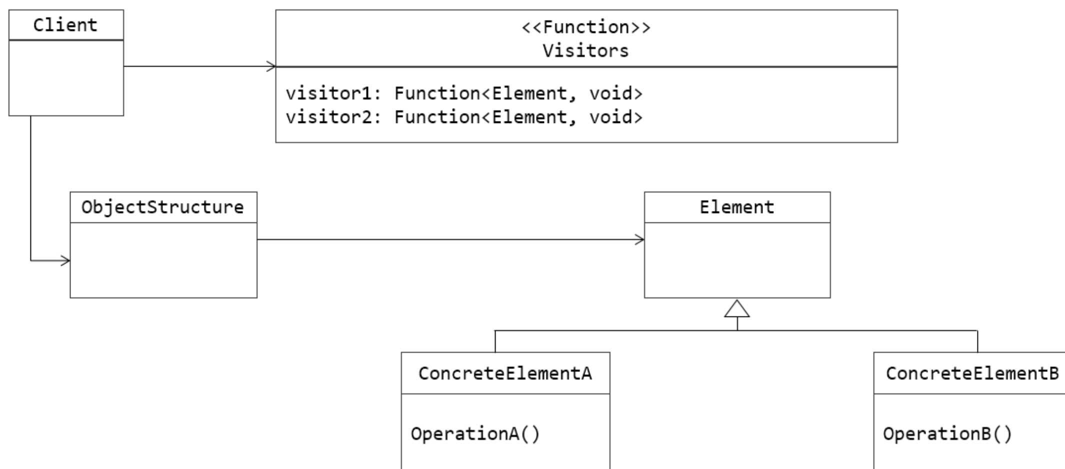

Figure 13: Modeling of visitor pattern with our function element.

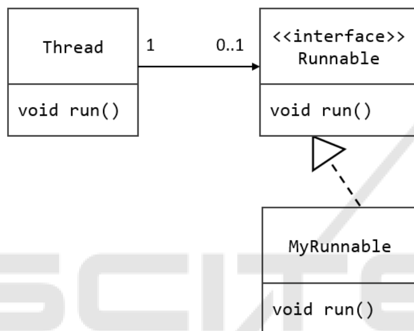

Figure 14: "Simple" version of the visitor in the Java runtime library (default).

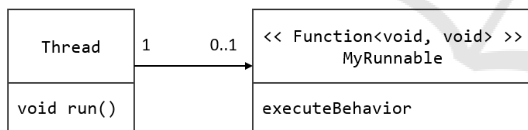

Figure 15: "Simple" version of the visitor in the Java runtime library (with function).

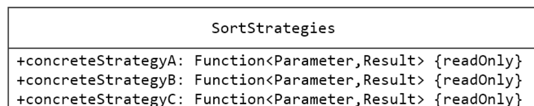

Figure 16: Modeling sorting function element with our UML meta-model.

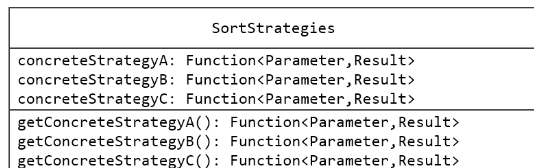

Figure 17: Modeling sorting function element with our UML meta-model (alternative).

model and activity transition graph. We suggest to apply our extension of UML and model the class model separately as shown in Figure 20. Our modeling approach would benefit the source code generation because we are close to the original EMOF.

7 CONCLUSION AND FUTURE WORK

In the last decade, lambda expressions and function references have become part of major OO languages, such as Java and C#. The modeling of these concepts has not evolved in the same way. This paper has shown how to extend UML class diagrams by a *function* element. The specification of the *function* element has been presented and evaluated. The evaluation has shown that behavioral modeling is made easier, thus making—at least in the cases of the strategy and visitor pattern—the modeling of design patterns less complex.

In the future, we evaluate benefits of our design in the area of Fog/Edge computing. As described by Hao et al. (Hao et al., 2017), tasks can move from device to device because of user movement or performance issues. Their suggested workflow approach could benefit from functional programming and probably from our modeling approach, too.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

```
public class SortStrategies {
  public final UMLFunc<List<Price>, List<Price>> pricesAsc = new UMLFunc<>();
  public final UMLFunc<List<Price>, List<Price>> pricesDesc = new UMLFunc<>();

  public SortStrategies() {
    pricesAsc.register(ArrayList.class, l -> lambda_expression);
    pricesDesc.register(ArrayList.class, l -> lambda_expression);
  }
}
```

Figure 18: Realization of the strategy pattern in Java for function element.

```
public class UMLFunc<T, R> implements Function<Object, R> {
  private final Map<Class<?>, Function> functionMap = new HashMap<>();

  public <T2 extends T> void register(Class<T2> clazz, Function<T2, R> function) {
    functionMap.put(clazz, function);
  }

  public R apply(Object o) {
    Function<Object, R> function = functionMap.get(o.getClass());
    return function != null ? function.apply(o) : null;
  }
}
```

Figure 19: Possible Java implementation of the function element.

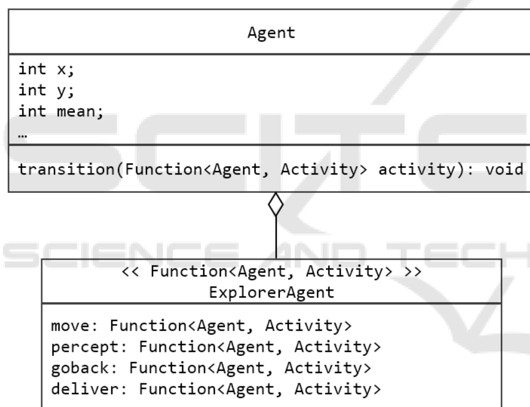

Figure 20: Application of new *Function* element based on (Bosse, 2016).

REFERENCES

Bosse, S. (2016). Mobile multi-agent systems for the internet-of-things and clouds using the javascript agent machine platform and machine learning as a service. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 244–253.

Dijkstra, E. (1984). The threats to computer science. EWD 898 (Delivered at the ACM 1984 South Central Regional Conference, November 16–18, Austin, Texas.).

Fusco, M. (2016a). Gang of Four Patterns in a Functional Light: Part 1. https://www.voxxed.com/2016/04/gang-four-patterns-functional-light-part-1/.

Fusco, M. (2016b). Gang of Four Patterns in a Functional Light: Part 2. https://www.voxxed.com/2016/05/gang-four-patterns-functional-light-part-2/.

Fusco, M. (2016c). Gang of Four Patterns in a Functional Light: Part 3. https://www.voxxed.com/2016/05/gang-four-patterns-functional-light-part-3/.

Fusco, M. (2016d). Gang of Four Patterns in a Functional Light: Part 4. https://www.voxxed.com/2016/05/gang-four-patterns-functional-light-part-4/.

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

Hao, Z., Novak, E., Yi, S., and Li, Q. (2017). Challenges and software architecture for fog computing. *IEEE Internet Computing*, 21(2):44–53.

Min, B.-K., Ko, M., Seo, Y., Kuk, S., and Kim, H. S. (2011). A UML metamodel for smart device application modeling based on Windows Phone 7 platform. In *TENCON 2011 - 2011 IEEE Region 10 Conference*, pages 201–205.

Object Management Group (OMG) (2016). OMG Meta Object Facility (MOF) Core 2.5.1 Specification. OMG Document Number formal/2016-11-01.

Object Modeling Group (2015). OMG Unified Modeling Language Version 2.5. http://www.omg.org/spec/UML/2.5/.