Restructering design patterns using functions in Java

Restructering design patterns using functions in Java

A. Bijlsma*[1], A.J.F. Kok†[1], H.J.M. Passier‡[1], H.J. Pootjes§[1] and S. Stuurman¶[1]

[1]Open Universiteit, Faculty of Management, Science and Technology, Department of Computer Science, Postbus 2960, 6401 DL Heerlen, The Netherlands

February 5, 2020

*lex.bijlsma@ou.nl
†arjan.kok@ou.nl
‡harrie.passier@ou.nl
§harold.pootjes@ou.nl
¶sylvia.stuurman@ou.nl

1 Introduction

Design patterns are standard solutions to common design problems. The famous Gang of Four (GoF) book describes twenty-three design patterns for the object-oriented (OO) paradigm [7]. Most of these patterns are based on the OO concepts delegation, inheritance, abstract class and interface.

Meanwhile, the functional paradigm has also become more popular. Besides pure functional languages, such as Haskell [11], more and more programming languages incorporate functional features and are in fact multi-paradigm languages. Examples are Scala [17] and JavaScript [18]. Java incorporates functional concepts from version 8, such as function objects as first class citizens and function composition, implemented by Java syntax constructs such as lambda expressions, functional interfaces and streams [12].
There are a number of examples on the world wibe web where object oriented design patterns are modified by applying functional features, for example [3, 4, 5, 6]. What is missing is a more thorough study of when and how it is useful to apply the functional features in design patterns.

In this report we investigate to what extent the solutions that OO design patterns offer can be replaced by functional features of Java, in such a way that the resulting solutions support more effectively the conceptual model underlying the original program design. We describe our research and results based on the strategy pattern. We also investigated other patterns, such as template method, visitor, decorator, and command. The specific results for these patterns can be found in [**?**]. Finally, we derive some rules of thumb to determine which patterns can be simplified and how this can be done.

UML class diagrams [2, 13] are helpful during the design and implementation of object oriented systems. Today, UML does not support functional features explicitly. It is a problem to show functional features as first-class citizens clearly in a UML class diagram. We shall propose one way of incorporating functional features into UML class diagrams.

This report In Section 2 we show several implementations of the strategy pattern and discuss the advantages and disadvantages of these implementations. Section 3 proposes an extension to UML to incorporate functions. Section 4 describes related work. The results are discussed in Section 5, which leads to conclusions and ideas for future work in Section 6.

2 The Strategy pattern

The strategy design pattern is intended to provide a way of selecting a strategy from a range of interchangeable strategies. This pattern defines several implementations of this strategy, and at runtime can be decided which implementation is used.

2.1 The standard object oriented approach

The GoF book shows an object oriented solution, see Figure 1 and Listing 1. Each concrete strategy is defined in a separate class that implements a common interface `Strategy` that defines the function(s) of the strategies. In Figure 1 the strategies define just one function: `execute` (types X and Y are not further specified). The strategy is used by the class `Context` in method `executeStrategy`. Which of the available strategies will be used, is set with method `setStrategy`, that is called with an instance of the required strategy.

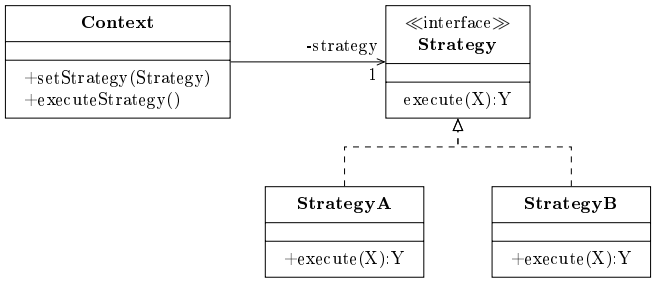

Figure 1: The strategy pattern, object oriented approach (Listing 1)

Listing 1: The strategy pattern, object oriented approach

```
public interface Strategy {
  public Y execute(X x);
}

public class StrategyA implements Strategy {

  public Y execute(X x) {
    // implementation for strategy A
  }
}

public class StrategyB implements Strategy {

  public Y execute(X x) {
    // implementation for strategy B
  }
}
```

```
public class Context {
  private Strategy strategy = new StrategyA();

  public void setStrategy(Strategy strategy) {
    this.strategy = strategy;
  }

  public void executeStrategy() {
    ...
    y = strategy.execute(x);
    ...
  }
}
```

A simple example how to apply the strategy pattern is given in Listing 2. Here, the instance of the concrete strategy is created directly. Usually this creation will be done with a factory.

Listing 2: Application of the object oriented strategy pattern

```
public static void main(String[] args) {
  Context context = new Context();
  context.setStrategy(new StrategyA());
  context.executeStrategy();
  context.setStrategy(new StrategyB());
  context.executeStrategy();
}
```

2.2 An alternative approach using an enumeration

With an enumeration it is possible to implement the equivalent of the strategy pattern without the class hierarchy, see Figure 2 and Listing 3. All different implementations of the strategy functions are defined in one enumeration `Strategy`. This enumeration replaces the class hierarchy of the object oriented approach. Each enumeration constant is coupled to one (or more) function(s), by implementing the strategy methods defined as abstract methods in the enumeration. In the given listing, each constant implements the strategy method `execute`. Class `Context` operates in the same way as in the object oriented version.

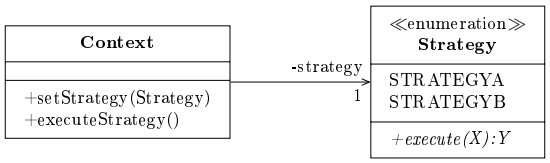

Figure 2: The strategy pattern, enumeration approach (Listing 3)

Listing 3: The strategy pattern, enumeration approach

```
public enum Strategy {

  STRATEGYA {
    public Y execute(X x) {
      // implementation of strategy A
    }
  },

  STRATEGYB {
    public Y execute(X x) {
      // implementation of strategy B
    }
  };

  public abstract Y execute(X x);
}

public class Context {
  private Strategy strategy = Strategy.STRATEGYA;

  public void setStrategy(Strategy strategy) {
    this.strategy = strategy;
  }

  public void executeStrategy() {
    ...
    y = strategy.execute(x);
    ...
  }
}
```

A simple example how to apply the strategy pattern is given in Listing 4. Note that now the client doesn't create instances of the strategies itself. So no factory is needed.

Listing 4: Application of the enumeration strategy pattern

```
public static void main(String[] args) {
  Context context = new Context();
  context.setStrategy(Strategy.STRATEGYA);
  context.executeStrategy();
  context.setStrategy(Strategy.STRATEGYB);
  context.executeStrategy();
}
```

2.3 An alternative approach with a functional interface

In the standard object oriented pattern the strategy is defined by an interface. When the methods of the interface are functions, then the interface can also be

used to store the implementation of these functions.

Figure 3 and Listing 5 show an implementation of the strategy pattern where the functions are stored in an interface. Each function in `Strategy` is a lambda function.

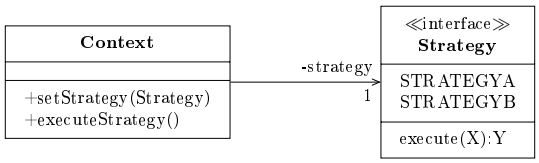

Figure 3: The strategy pattern, functions defined in a functional interface (listing 5)

Listing 5: The strategy pattern, functions defined in interface

```
public class Context {
  private Strategy strategy = Strategy.STRATEGYA;

  public void setStrategy(Strategy strategy) {
    this.strategy = strategy;
  }

  public void executeStrategy() {
    ...
    y = strategy.execute(x);
    ...
  }
}

public interface Strategy {
   Y execute(X x);

   public Strategy STRATEGYA = (x) -> ... // a lambda expression
   public Strategy STRATEGYB = (x) -> ... // another lambda expression
}
```

A simple example how to apply the strategy pattern is given in Listing 6.

Listing 6: Application of strategy pattern with functions from an interface

```
public static void main(String[] args) {
  Context context = new Context();
  context.setStrategy(Strategy.STRATEGYA);
  context.executeStrategy();
  context.setStrategy(Strategy.STRATEGYB);
  context.executeStrategy();
}
```

However, this solution does not restrict the strategies to be used to those defined in given interface. All methods or lambda expressions that match the interface `Strategy` can be passed to method `setStrategy`, wherever they are defined, see Listing 7. Therefore, it is a very flexible solution, but there is not much control of what the strategies will do.

Listing 7: Application of the functional strategy pattern

```
public static void main(String[] args) {
  Context context = new Context();
  context.setStrategy(Strategy.STRATEGYA);
  context.executeStrategy();
  context.setStrategy(Strategy.STRATEGYB);
  context.executeStrategy();

  // class C contains method that matches interface Strategy
  context.setStrategy(C::method);
  context.executeStrategy();
  context.setStrategy((x)->doSomethingCompletelyDifferent(x));
  context.executeStrategy();
}
```

Another disadvantage of this approach arises when a strategy consists of two or more methods. For example, when the strategy is defined by methods `method1` and `method2`, these must be defined in two seperate interfaces, as a functional interface can only contain one method definition. Furthermore, all combinations of implementations of `method1` and `method2` are possible. That can not be prevented. This is in conflict with the intent of the strategy pattern, that should enforce only certain combinations of these methods.

2.4 Discussion of the approaches

The object oriented and enumeration approaches meet the intent of the strategy pattern: define a family of algorithms and make them interchangeable. The interface approach does not limit the possible strategies, and therefore does not completely meet this intent. This interface approach will not be considered further.

The main difference between the two remaining approaches is where the functions are defined: each strategy in its own class for the object oriented approach or all strategies in one enumeration for the enumeration approach.

The advantage of the object oriented approach is that adding a new strategy only means adding a new class for this new strategy. Existing classes are not modified. Adding a new strategy in the enumeration approach means extending the existing enumeration (`Strategy`). However, this enumeration is only extended, existing code is not modified. In all cases the extension can be added without modifying existing code. Therefore, both approaches satisfy the Open-Closed principle [14].

The enumeration approach simplifies the class structure, i.e. the subclasses of the strategy have been removed. The cost of this simplification of the class structure is an increased size of the enumeration or the class that contains the strategies, as all implementations of strategies are now collected in this enumeration or class. Therefore, the enumeration approach seems to be most applicable when the implementations of the strategies are simple, i.e. exist of a limited number of lines of code.

There is a difference in the exact functionality between the different approaches. A concrete strategy in the enumeration has singleton behavior: all users of a strategy in an application use the same object. In the object oriented approach an application can create and use several instances of the same strategy. This difference only shows when the strategies store states. When the strategies contain pure functions, this difference can be ignored.

From the UML class diagrams of Figure 1, it is directly clear that it represents a strategy pattern and which variations of the strategy exist. The pattern is not explicitly present in the UML class diagram of the enumeration approach in Figure 2. Only the names of the strategies are directly visible, but not the functions for each strategy. We will discuss a proposal to extend UML for enumerations in Section 3.

In some applications the subclasses of `Strategy` in the object oriented approach need to store information (state) as attributes. The functions in the enumeration approach do not have attributes to store state, as this approach uses pure functions. To overcome this problem, the Context can manage the state information and pass this information to the functions as parameters.

When each strategy needs another type of state information, so each strategy needs its own class, then the object oriented approach is preferred over the other approaches. The advantage of the other approaches is eliminated, as the number of state objects equals the number of subclasses, so no reduction of classes is achieved.

3 UML extension: a proposal

In Section 2 we gave a simpler solution to the problem underlying the GoF Strategy pattern, using enumerations and functional abstraction. However, these solutions are far harder to describe in UML than the classical approach. The essence of the pattern, in our view, is the possibility of a dynamic choice between statically defined alternatives. In the object-oriented style of Figure 1, this is clearly visible because of the dynamically mutable association from `Context` to `Strategy`, as opposed to the statically fixed implementation relationship between interface `Strategy` and concrete classes `StrategyA` and `StrategyB`.

In Figure 2 the concrete strategies are no longer visible except as untyped constants in the enumeration. This is because UML is entirely geared to relations between classes, and in the simplified enum-style solution the concrete strategies are no longer represented as classes. They are, in fact, first-class functions – not methods. The only way to represent a first-class function in UML is to view this as an object implementing a functional interface. However, it is very awkward to have to show this library interface in the diagram every time a function is used.

This situation suggests that we would like to extend UML with a dedicated notation for such first-class functions. Then in Figure 2 the enumeration elements could be explicitly linked to the functions they represent.

In order to remain as close as possible to standard UML, we propose to use a rectangle with rounded left and right sides: a so-called 'capsule shape'. These do not play a role in normal class diagrams, but the shape is used in activity diagrams to denote an activity. This does not seem to clash strongly with the proposed use as a notation for stand-alone functions. Using this shape to denote the functions associated with the enumeration elements, Figure 2 may be replaced by Figure 4. We claim that this notation makes it easier to see the dynamic choice between statically defined alternatives, which was what we set out to do.

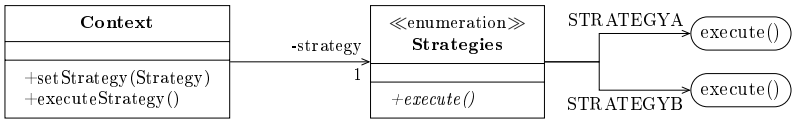

Figure 4: The strategy pattern, enum style (Listing 3)

Being able to model a solution is beneficial for students, because it allows them to think about a solution in abstract terms without having to attend every detail [1]. Furthermore, a UML diagram is beneficial in communication with domain experts, because diagrams are far more easy to understand than code.

4 Related work

It has been observed many times before that design patterns reflect a lack of features in programming languages. The GoF patterns [7] correspond to the set of features current in mainstream object-oriented languages such as C++ and Java around the time of the book's publication. Sullivan [16] showed that using a more permissive object-oriented language would make some design patterns disappear. Hannemann and Kiczales [8] explored expressing the GoF patterns in AspectJ, with the result that in many cases the core part of the implementation could be abstracted into reusable code, thus creating a component rather than a pattern.

An early proposal to exploit the new Java functional features in the context of design patterns was made by Fusco [3, 4, 5, 6]. However, his approach is entirely code-based and rather ad hoc: it provides one with several examples where existing code is cleaned up and simplified, but omits any consistent methodology and does not aid at all in the design phase.

The recent work of Heinzl and Schreibmann [9] does share our ambition for the early introduction of lambda expressions, and proposes an extension to UML to facilitate the design process accordingly. However, their choice of a class symbol to represent a function is confusing: a function is not a class but an object of type `Function<P, R>`. Moreover, they seem to use the same multiplicity notation for referring both to n objects and to a single object with n attributes. Finally, their notation blurs the essence of some design patterns: the Strategy pattern, for instance, is about making a dynamic choice from a repertoire of algorithms. In the description of Heinzl and Schreibmann all the algorithms are present simultaneously as attributes, and the dynamic aspect vanishes from the design.

5 Discussion

5.1 Generalization of the method

Our goal was to simplify the class structure of design patterns to bring the design more into line with the conceptual model. We have done this by removing the inheritance structure and replace is with an enumeration or with function interfaces:

Enumeration. The functions in the concrete subclasses in the object oriented approach are collected into one enumeration, and stored labeled with an enumeration constant.

Functional interfaces. The functions in the concrete subclasses in the object oriented approach are collected in one interface.

We discussed that the interface approach has several disadvantages. I t is not enforced that the functions are defined in one place, i.e. one interface. And when the design pattern is defined by two ore more functions, we cannot restrict to the desired combinations of implementations.

The enumeration approach is a good alternative for the object oriented approach, as long as the implementions of the functions are simple, and are free of state. It is, however, possible to use state by transferring it by a parameter object, but this makes the interface solution more complex, so than in most cases the object oriented approach is preferred.

5.2 Applicability for other patterns

Our approach is feasible when:

- Methods are pure functions, that is when they do not rely on attributes (state). In cases where the methods rely on very simple (only a few attributes of simple types), then this state can be realized by passing state as parameter. In these cases the caller of the functions is responsible for managing the state.
- The redesigned design pattern should not become more complex than the original object oriented pattern and should support the conceptual way of thinking.

Given these conditions, for the three categories of patterns, behavioral, creational, structural, we will conclude with some general statements about the applicabiltiy of our approaches:

- Behavorial patterns are the most likely candidates for applying the enumaration approach. Behavior is often defined by algorithms, and thus by pure functions.
- The creational patterns can be reconstructed using functions, as long as no state is needed to construct the final object structure.

- The enumeration and interface approach are less applicable for structural patterns, where structure is added in terms of attributes, as these approaches do not offer an easy way to store attributes. When the structure is only defined by pure functions, then for some of the structural patterns our approaches are both applicable and simplify the application. For example, in the famous coffee example [?] for the decorator pattern our approaches are applicable, because the added functionality consists of pure functions, as for example the computation of the price of a cup of coffee.

6 Conclusions and future work

We investigated the use of functional features for several design patterns. Our goal was to simplify the class structure of design patterns to bring the design more into line with the conceptual model. We have done this by removing the inheritance structure and replacing it with an enumeration. The functions in the concrete subclasses in the object oriented approach are collected into one enumeration, and stored labeled with an enumeration constant.
Our approach is feasible when:

- Methods are pure functions, i.e. when they do not rely on attributes (state). In cases where the methods rely on very simple state (only a few attributes of simple types), then this state can be realized by passing state as parameter. In these cases the caller of the functions is responsible for managing the state.
- Methods are of limited complexity and size. Otherwise, for example, the enumeration will become very large.
- The redesigned design pattern should not become more complex than the original object oriented pattern and should support the conceptual way of thinking.

Best suited are the patterns classified as behavioral, for example Strategy and Template Method, as they deal with algorithms (functions). However, some of the behavioral patterns are less suitable, for example Command and Visitor. Structural patterns deal with structure, and therefore are less suited. In some special cases, however, also structural patterns can use our approach, for example the Decorator pattern when the decorations are pure functions.

6.1 Future work

The question which design patterns form a suitable candidate for improvement through functional features does not seem to allow of a simple answer: as argued in the previous section, the dichotomy between behavioral and structural patterns comes close to providing a criterion, but Visitor and Decorator are notable counterexamples. Ideally one would wish for an objective criterion pointing to the cases where our approach adds value. One avenue to explore in this direction would be the application of various quality metrics [10]. However, it is worth pointing out that design patterns do not improve all quality aspects: they have a purpose, for instance contributing to flexibility for certain types of changes, but often do so by increasing the number of classes or adding a level of indirection, all of which would deteriorate other quality metrics.

A different approach to analyzing design patterns was offered by Smith [15], who considered them as compositions of much simpler programming ideas that cannot be decomposed further. The structure of such compositions provides an indication of conceptual complexity for each classical design pattern and also

for our alternative versions: this might lead to an objective criterion of the kind we are looking for.

A final remark that must be made is that the possible solutions considered here are constrained by what is possible within present versions of Java. Related languages such as Scala would lead to different choices. Therefore it would be worth while to investigate what language features would be necessary for even simpler versions of design patterns. For example, the Singleton pattern disappears entirely in Scala because the language offers the possibility of defining individual objects not belonging to any class.

References

[1] Vladislav Georgiev Alfredov. How programming languages affect design patterns, a comparative study of programming languages and design patterns. Master's thesis, Department of Informatics, University of Oslo, Autumn 2016.

[2] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[3] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.

[4] M. Fusco. Gang of Four Patterns in a Functional Light: Part 1. `https://www.voxxed.com/2016/04/gang-fourpatterns-functional-light-part-1/`, 2016.

[5] M. Fusco. Gang of Four Patterns in a Functional Light: Part 2. `https://www.voxxed.com/2016/05/gang-fourpatterns-functional-light-part-2/`, 2016.

[6] M. Fusco. Gang of four patterns in a functional light: Part 3. `https://www.voxxed.com/2016/05/gang-fourpatterns-functional-light-part-3/`, 2016.

[7] M. Fusco. Gang of four patterns in a functional light: Part 4. `https://www.voxxed.com/2016/05/gang-fourpatterns-functional-light-part-4/`, 2016.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Reading, MA, USA, 1995.

[9] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. *SIGPLAN Not.*, 37(11):161–173, November 2002.

[10] Steffen Heinzl and Vitaliy Schreibmann. Function references as first class citizens in uml class modeling. In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*, pages 335–342. INSTICC, SciTePress, 2018.

[11] Nien-Lin Hsueh, Peng-Hua Chu, and William Chu. A quantitative approach for evaluating the quality of design patterns. *Journal of Systems and Software*, 81(8):1430 – 1439, 2008.

[12] Graham Hutton. *Programming in Haskell*. Cambridge University Press, New York, NY, USA, 2nd edition, 2016.

[13] Gosling J., Joy B., Steele G., Bracha G., and Buckley A. *The Java Language Specification (3rd edn)*. Addison-Wesley, 2014.

[14] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2009.

[15] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall PTR: Upper Saddle River, NJ, U.S.A., 1997.

[16] Jason McC. Smith. *Elemental Design Patterns*. Addison-Wesley Professional, 2012.

[17] Gregory T Sullivan. Advanced programming language features for executable design patterns: Better patterns through reflection. Lab memo AIM-2002-005, MIT Artificial Intelligence Laboratory, 2002.

[18] B.P. Upadhyaya. *Programming with Scala: Language Exploration*. Undergraduate Topics in Computer Science. Springer International Publishing, 2017.

[19] Nicholas C. Zakas. *Understanding ECMAScript 6: The Definitive Guide for JavaScript Developers*. No Starch Press, San Francisco, CA, USA, 1st edition, 2016.