Eliminating OO Patterns by Java Functional Features

An explorative study

Technical Report (TR-OU-INF-2019-01)

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*

October 30, 2019

*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 [9]. 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 [13], more and more programming languages incorporate functional features and are in fact multi-paradigm languages. Examples are Scala [19] and JavaScript [21]. 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 [14].

A disadvantage of the classical OO design patterns is that they do not always fit domain concepts when applied in concrete situations. For an example, look at the Strategy pattern, where a context object maintains a reference to a strategy defining an interface common to all possible concrete strategies. The strategy can be a real interface or an (abstract) class. Each concrete strategy requires a separate class in Java, even when the concrete strategies are in fact pure functions. Therefore, the conceptual model of a strategy, i.e. define a family of algorithms and choose one to use, becomes muddied by compound concepts as inheritance and complex class/object structures due to the limitations of the OO language syntax constructs.

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. It becomes clear that certain patterns can be simplified using functional features where we often use enum types as containers to hold functions together. We investigate the patterns that are concerned with algorithms. So, behavioral patterns are most likely to be investigated. However, we do not exclude the other (creational and structural) patterns. Finally, we derive some rules of thumb to determine which patterns can be simplified and how this can be done.

UML class diagrams [2, 15] 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.

Contributions. In this report, we have the following contributions:

- For the patterns Strategy, Template Method, Command, Decorator and Vistor we discuss and present alternative structures using Java's functional features and investigate the advantages and disadvantages of applying functional features to these patterns.

- We derive rules of thumb to determine when a pattern can be simplified using functional features and how this can be done.
- We present a proposal for how functional features can be depicted in UML design class diagrams.

Remark. The symbol △ indicates the end of a remark and the resumption of normal text. Likewise, the symbol □ indicates the end of an short delimited example. △

Remark. We limit ourselves to the programming language Java version 11 and UML version 2.5. △

This report. Section 2 describes briefly the functional features of Java we use in this report. In Section 3, we start with analyzing the Strategy pattern which seems an obvious candidate for the application of functional features. Next, the patterns Template Method, Command, Decorator and Visitor follow in the Sections 4, 5, 6 and 7. In each of these sections, we start with presenting an example using the standard pattern as described in the GoF book. After that, we try to simplify the pattern using functional features. We analyze the advantages and disadvantages in terms of reduction of complexity and increase of flexibility. Section 8 proposes how functional features can be incorporated into UML class diagrams. We describe related work in Section 9. Finally, in we discuss our results, present our rules of thumb to determine when and how functional features can be applied generally to simplify design patterns, draw our conclusions and describe future work.

2 Preliminaries

Functional programming in Java makes use of lambda expressions, function objects, functional interfaces and streams [20]. These topics are briefly described in the following subsections.

2.1 Lambda expressions in Java

A lambda expression in Java resembles syntactically the standard notation, i.e. *parameter list* $\rightarrow$ *function body*. One argument can be listed in or without parentheses. Thus, `x -> x+1` as well as `(x) -> x+1` are both correct Java expressions. No argument is represented by an empty parameter list, for example `() -> System.out.println("Hello world")`. For multiple parameters, parentheses are required, as in `(x,y,z) -> x+y+z`.

If an expression contains a single statement, no curly braces enclosing the expression's body are needed. Curly braces are required when the body consists of two or more statements.

There is no need to declare the type of a parameter, i.e. the compiler can infer the type of the parameters.

When the body has one single expression, the compiler automatically returns the resulting value (without the need of a return statement). A return is needed in cases of multiple statements.

2.2 Functional interfaces in Java

A Java functional interface is annotated by `@FunctionalInterface` and has exactly one abstract method. The number of arguments is free as is the type of the arguments. Listing 1 shows an example.

Listing 1: Example of a Java functional interface

```
@FunctionalInterface
public interface IFunction {
  int doFunction(int a, int b);
}
```

It is common practice to get instances of this interface using a factory as shown in Listing 2. Factory `FunctionFactory` provides two instances of `doFunction`: a function that adds two integer and a function that multiplies two integers. Notice that in both cases an anonymous function in the form of a lambda expression is returned. The compiler takes care of bounding both function expressions to the function's name `doFunction` of the functional interface.

Listing 2: Use of a factory

```
public class FunctionFactory {

  public static IFunction getFunctionAdd() {
    return (a, b) ->  a + b;
  }

  public static IFunction getFunctionMul() {
    return (a, b) -> a * b;
  }
}

public class Main {
  public static void main(String[] args) {
    IFunction mul = FunctionFactory.getFunctionMul();
    System.out.println(mul.doFunction(3, 7));
  }
}
```

An alternative for specifying and implementing your own functional interfaces, is the use of Java's predefined functional interfaces, see Table 1. An example of a predefined functional interface is `BinaryOperator<Integer>` which accepts a lambda expression that takes two integers as arguments and returns an integer as result value. To invoke the lambda expression, we have to use BinaryOperator's method `apply`. An example is given in Listing 3.

Listing 3: Use of the predefined interface `BinaryOperator<Integer>`

```
import java.util.function.BinaryOperator;
public class FunctionFactory {

  public static BinaryOperator<Integer> getFunctionAdd() {
    return (a, b) ->  a + b;
  }

  public static BinaryOperator<Integer> getFunctionMul() {
    return (a, b) -> a * b;
  }
}

public class Main {
  public static void main(String[] args) {
    BinaryOperator<Integer> mul = FunctionFactory.getFunctionMul();
    System.out.println(mul.apply(3, 7));
  }
}
```

Table 1: Java's predefined functional interfaces

Functional Interface	Parameter Types	Return Type	Abstract Method	Other Methods
`Runnable`	none	**`void`**	`run`	
`Supplier<T>`	none	`T`	`get`	
`Consumer<T>`	`T`	**`void`**	`accept`	`andThen`
`BiConsumer<T, U>`	`T, U`	**`void`**	`accept`	`andThen`
`Function<T, R>`	`T`	`R`	`apply`	`compose`, `andThen`, `identity`
`BiFunction<T, U, R>`	`T, U`	`R`	`apply`	`andThen`
`UnaryOperator<T>`	`T`	`T`	`apply`	`compose`, `andThen`, `identity`
`BinaryOperator<T>`	`T, T`	`T`	`apply`	`andThen`
`Predicate<T>`	`T`	**`boolean`**	`test`	`and`, `or`, `negate`, `isEqual`
`BiPredicate<T, U>`	`T, U`	**`boolean`**	`test`	`and`, `or`, `negate`

2.3 Streams

Streams can be used as a functional alternative for collections. The standard workflow is to convert a collection into a stream of type `Stream`. The `Stream` interface contains a series of operations each of which corresponds to a known operation, for example map and filter, that performs on a collection. Using these operations, the stream can be manipulated in a functional manner by composing a number of functions into a pipeline. After manipulation, the stream can be transformed back into a collection.

Stream operations are divided into *intermediate* operations and *terminal* operations. Intermediate operations transform a stream into another stream. An example is `filter(Predicate)` that returns a substream consisting of the elements that match given predicate. Predicate is a lambda function that takes one element and returns true or false. Other examples are `map()` and `sorted()`. Terminal operations produce a final result or side effect. An example is `count()` resulting in a number, i.e. the number of elements in the stream. Other examples are `forEach()`, `reduce()`, `min()`, `max()`, and `sum()`. Most of these operations take a lambda expression as argument. Finally, there are operations to convert a collection instance into a stream back and forth. An example is `stream()`.

Manipulating a stream is often performed using a pipeline. A pipeline con-

sists of a source (which might be an array, a collection, a generator function, an I/O channel, et cetera), followed by zero or more intermediate operations (transforming a stream into another manipulated stream), possibly followed by a terminal operation (producing a final result or side effect).

Suppose we have a number of integers as input and we want to display the square of the even numbers only. Listing 4 shows the creation of a stream and the pipeline.

Listing 4: Creation of a stream and pipeline

```
IntStream stream = IntStream.of(7, 8, 9, 10, 11, 12);
stream.filter(e -> e % 2 == 0)
      .map(e -> e * e)
      .forEach(e -> System.out.println(e));
```

When in a lambda expression an already defined method is invoked, we can use a method reference. In listing 4 instead of `e -> System.out.println(e)` we can write `System.out::println` as parameter of the `forEach()` method.

Furthermore, instead of using nameless lambda expression we can give them a name so they can be reused in other expressions. Listing 5 shows an example.

Listing 5: Method reference and named lambda expression

```
Intstream stream = IntStream.of(7, 8, 9, 10, 11, 12);
IntPredicate isEven = e -> e % 2 == 0;
IntUnaryOperator square =  e -> e * e;
stream.filter(isEven)
      .map(square)
      .forEach(System.out::println);
```

2.4 Function composition

Several of Java's functional interfaces of Table 1 define methods `compose()` and `andThen()` to combine several instances of the interface into a complex function. Listing 6 shows a simple example.

Listing 6: Function composition (I)

```
Function<Integer, Integer> plus1 = i -> i + 1;
Function<Integer, Integer> doubleIt = i -> i * 2;
Function<Integer, Integer> composition = plus1.andThen(doubleIt);
System.out.println(composition.apply(5)); // result is 12
```

Two instances of functional interface `Function` are composed into a new function. First, `plus1()` is applied and then `doubleIt()`. If we had used `compose()`

instead of `andThen()` the order of application had been reversed. Because the result type of this composition is `Function`, we can compose an even more complex function, as shown in Listing 7.

Listing 7: Function composition (II)

```
Function<Integer, Integer> composition2 =
         composition.andThen(plus1).andThen(doubleIt);
System.out.println(composition2.apply(5));   // result is 26
```

2.5 Constants, enumerations and lambda expressions

There are several ways to refer to a lambda expression. Because in most cases the lambda expressions are known at compile time, we can use constants, i.e. static final variables. Listing 8 shows and example.

Listing 8: Referring to a lambda expression

```
public static final BinaryOperator<Integer> DIRECTMUL =
                                              (x, y) -> x * y;
```

An alternative is to use an enumeration. Enumerations are suitable to hold a number of related functions, where each function is labeled with a enumeration constant. An enumeration (enum type) can be regarded as a special class with attributes, methods and a constructor. If an enumeration has an abstract method, then each constant of the enumeration must implement this method. The implementation can be a normal method without a lambda expression. An example is shown in Listing 9.

Listing 9: The enumeration as holder of methods

```
public enum Multiply {

  DIRECTMUL {
    public Integer apply(Integer a, Integer b) {
      return a * b;
    }
  },

  OTHERMUL {
    public Integer apply(Integer a, Integer b) {
      return otherMul(a, b);
    }
  }

  public abstract Integer apply(Integer a, Integer b);
```

```
  private static Integer otherMul(Integer a, Integer b) {
    // other implementation of multiplication
  }
}
```

We can perform multiplications by calling the method apply on the enumeration constant as in Listing 10.

Listing 10: Example of usage of enumeration

```
System.out.println(Multiply.DIRECTMUL.apply(3, 4));
System.out.println(Multiply.OTHERMUL.apply(3, 4));
```

We can extract the definition of the function `apply` to a (functional) interface. We can define our own interface. However, in this case we can use the standard interface BinaryOperator, as it defines the required signature for the apply method. Now, the enumeration can implement this interface, see Listing 11.

Listing 11: The enumeration as holder of methods implementing an interface

```
public enum Multiply implements BinaryOperator<Integer> {
  DIRECTMUL {
    public Integer apply(Integer a, Integer b) {
      return a * b;
    }
  },
  OTHERMUL {
    public Integer apply(Integer a, Integer b) {
      return otherMul(a, b);
    }
  };

  private static Integer otherMul(Integer a, Integer b) {
    // other implementation of multiplication
  }
}
```

An advantage of using a standard interface over a self-defined interface is that the extra functionality of the standard functional interfaces can direclty be used, for example for function composition. An example is given in Listing 12. The result of the multiplication is incremented by 1.

Listing 12: Example of usage of an enumeration with function composition

```
Function<Integer, Integer> plus1 = i -> i + 1;
System.out.println(Multiply2.DIRECTMUL.andThen(plus1).apply(3, 4));
```

We can define a direct relation between an enumeration constant and a lambda expression as function. An attribute will store the lambda expression, and this attribute gets is value (i.e. the lambda expression) using a constructor. The type of the attribute must be an (functional) interface. The exact type depends on the lambda expression, i.e. its number and type of parameters and its type of return value. An example is given in Listing 10. The lambda expressions used have two integer parameters and an integer return value. Therefore, functional interface `BinaryOperator<Integer>` is a suitable type, see Table 1. We can call method apply in the same way as before (Listing 10).

Listing 13: Extending the enumeration

```
public enum Multiply implements BinaryOperator<Integer> {

  DIRECTMUL((a, b) -> a * b),
  OTHERMUL( (a, b) -> otherMul(a, b));

  private BinaryOperator<Integer> multiplier;

  Multiply(BinaryOperator<Integer> multiplier){
    this.multiplier = multiplier;
  }

  private static Integer otherMul(Integer a, Integer b) {
    // other implementation of multiplication
  }

  public Integer apply(Integer a,Integer b) {
    return multiplier.apply(a, b);
  }
}
```

The introduction of the lambda expressions in the enumeration makes the code more complex, because an attribute and a constructor are introduced. Therefore, when a enumeration is used as holder of functions, the versions of Listings 9 and 11 are most suitable.

An alternative is to use a class or an interface as holder of functions. Listing 14 shows an interface containing functions. We can call method `apply` of the functions in the same way as before (Listings 10 and 12).

Listing 14: Interface containing functions

```
public interface Multiply {
```

```
  BinaryOperator<Integer> DIRECTMUL = (a , b) -> a + b;
  BinaryOperator<Integer> OTHERMUL = (a, b) -> otherMul(a, b);

  private static Integer otherMul(Integer a, Integer b) {
    // other implementation of multiplication
  }
}
```

This version is the most concise.

3 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.

3.1 The standard object oriented approach

The GoF book shows an object oriented solution, see Figure 1 and Listing 15. 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.

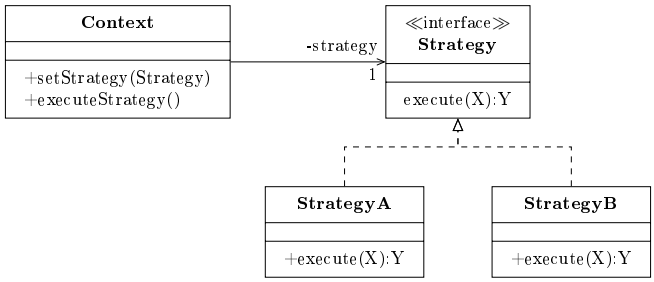

Figure 1: The strategy pattern, object oriented approach (Listing 15)

Listing 15: 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;

  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 16. Here, the instance of the concrete strategy is created directly. Usually this creation will be done with a factory.

Listing 16: 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();
}
```

3.2 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 17. All different implementations of the strategy functions are defined in one enumeration `Strategies`. 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.

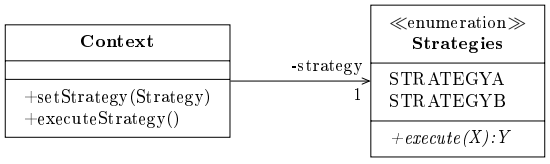

Figure 2: The strategy pattern, enumeration approach (Listing 17)

Listing 17: The strategy pattern, enumeration approach

```
public enum Strategies {

  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 Strategies strategy;

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

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

A simple example how to apply the strategy pattern is given in Listing 18. Note that now the client doesn't create instances of the strategies itself. So no factory is needed.

Listing 18: Application of the enumeration strategy pattern

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

3.3 An alternative approach with an interface

Section 2.5 showed that an interface (or class) also can be used to store functions in a compact way. Figure 3 and Listing 19 show an implementation of the

strategy pattern where the functions are stored in an interface. Each function in `Strategies` is a lambda function.

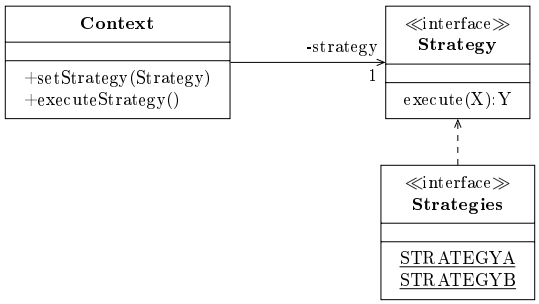

Figure 3: The strategy pattern, functions defined in an interface (listing 19)

Listing 19: The strategy pattern, functions defined in interface

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

public class Context {
  private Strategy strategy;

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

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

public interface Strategies {
   STRATEGYA = (x) -> ...  // a lambda expression using parameter x
   STRATEGYB = (x) -> ...  // another lambda expression using x
}
```

A simple example how to apply the strategy pattern is given in Listing 20.

Listing 20: Application of strategy pattern with functions from an interface

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

```
  context.setStrategy(Strategies.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 21. Therefore, it is a very flexible solution, but there is not much control of what the strategies will do.

Listing 21: Application of the functional strategy pattern

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

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

3.4 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 [1].

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 (`Strategies`). 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 [16].

[1] For the other patterns in this report the interface approach will have the same disadvantage. Therefore, we will not investigate this approach for the other patterns.

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.

Remark. 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. △

Remark. 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 8. △

Remark. 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. △

4 The Template Method pattern

The template method design pattern is intended to vary part(s) of the implementation of an algorithm without changing the structure of the algorithm (where the strategy pattern varies the complete algorithm).

4.1 The standard object oriented approach

The GoF book shows an object oriented solution, see Figure 4 and Listing 22. The skeleton of an algorithm (`templateMethod`) is defined in a class together with some abstract 'hook' methods (`primitiveOperation1`, `primitiveOperation2`). These hook methods are used within the algorithm, but the implementations of these methods are deferred to concrete subclasses (`ClassA` and `ClassB`). To guarantee that the skeleton of the algorithm can not be overridden in the subclasses, method `templateMethod` is usually declared final.

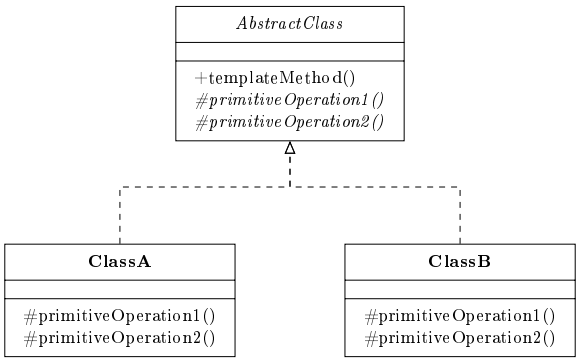

Figure 4: The template method pattern, object oriented approach (Listing 22)

Listing 22: The template method pattern, object oriented approach

```
public abstract class AbstractClass {

  public final void templateMethod() {
    ...
    primitiveOperation1();
    ...
    primitiveOperation2();
    ...
  }

  protected abstract void primitiveOperation1();
  protected abstract void primitiveOperation2();
}

public class ClassA extends AbstractClass {
```

```
  protected void primitiveOperation1() {
    // implementation
  }

  protected void primitiveOperation2() {
    // implementation
  }
}

public class ClassB extends AbstractClass {

  protected void primitiveOperation1() {
    // implementation
  }

  protected void primitiveOperation2() {
    // implementation
  }
}
```

A simple example how to apply the template method pattern is given in Listing 23. Here, the instance of the concrete classes are created directly. Usually this will be done with a factory.

Listing 23: Application of the object oriented template method pattern

```
public static void main(String[] args) {
  AbstractClass class1 = new ClassA();
  class1.templateMethod();
  AbstractClass class2 = new ClassB();
  class2.templateMethod();
}
```

4.2 An alternative approach using an enumeration

With an enumeration it is possible to implement the equivalent of the template method pattern without the class hierarchy, see Figure 5 and Listing 24. The primitive (hook) operations to be used in the template method (`templateMethod`) are defined in the enumeration `Hooks`. By assigning an enumeration constant to the template class, the hook operations defined by given enumeration constant are applied in `templateMethod`.

Listing 24: The template pattern, enumeration approach

```
public class TemplateClass {
  private Hooks hooks;
```

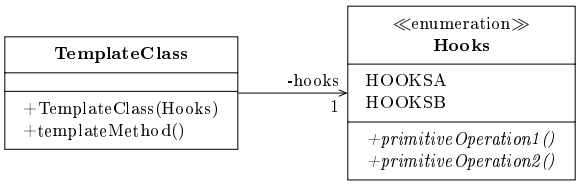

Figure 5: The template pattern, enumeration approach (Listing 24)

```
  public TemplateClass(Hooks hooks) {
    this.hooks = hooks;
  }

  public final void templateMethod() {
    ...
    hooks.primitiveOperation1();
    ...
    hooks.primitiveOperation2();
    ...
  }
}

public enum Hooks {

  HOOKSA {

    public void primitiveOperation1() {
      // implementation
    }

    public void primitiveOperation2() {
      // implementation
    }
  },

  HOOKSB {

    public void primitiveOperation1() {
      // implementation
    }

    public void primitiveOperation2() {
      // implementation
    }
  };

  public abstract void primitiveOperation1();
  public abstract void primitiveOperation2();
}
```

A simple example how to apply the template method pattern is given in Listing 25. Note that now the client doesn't create instances of the concrete

classes itself as in the object oriented approach. So no factory is needed.

Listing 25: Application of the enumeration template method pattern

```
public static void main(String[] args) {
  TemplateClass class1 = new TemplateClass(Hooks.HOOKSA);
  class1.templateMethod();
  TemplateClass class2 = new TemplateClass(Hooks.HOOKSB);
  class2.templateMethod();
}
```

4.3 Discussion of the approaches

The advantages and disadvantages are the same as for the Strategy pattern, summarizing:

- Both approaches satisfy the intent of the Template pattern.
- Both approaches satisfy the Open-Closed principle.
- In the enumeration approach the class structure is simplified.
- The enumeration approach seems to be most applicable when the implementations of the template methods are simple, i.e. exist of limited number of lines of code.
- When state information is needed in the enumeration approach, this information has to be passed as parameters tot the functions.

5 The Command pattern

The command design pattern is intended to "encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations" [9]. The command pattern decouples the object that invokes the operation from the object that has the knowledge to execute it.

5.1 The standard object oriented approach

The standard object oriented solution is shown in Figure 6 and Listing 26. Concrete commands (here of classes `CommandA` and `CommandB`) define a binding between a receiver object and an action on this object, that is performed in the execute method of the command. Requests are carried out by the invoker object, by delegating the responsibility to a command object.

In this example the invoker stores one command at a time. In many examples of the command pattern the invoker stores several commands in a collection. In those examples the method `execute` of the invoker executes all stored commands or first selects one of the commands to be executed.

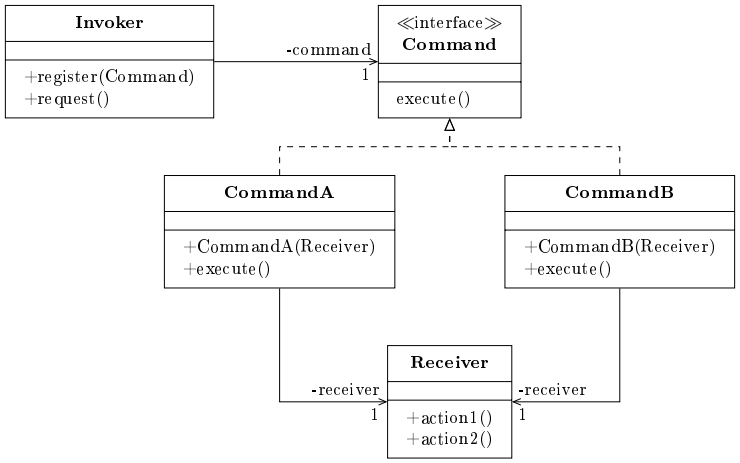

Figure 6: The command pattern, object oriented approach (Listing 26)

Listing 26: The command pattern, object oriented approach

```
public class Invoker {
  private Command command;

  public void register(Command command) {
    this.command = command;
  }
```

```
  public void request() {
    command.execute();
  }
}

public interface Command {
  void execute();
}

public class CommandA implements Command {
  private Receiver receiver;

  public CommandA(Receiver receiver) {
    this.receiver = receiver;
  }

  public void execute() {
    receiver.action1();
  }
}

public class CommandB implements Command {
  private Receiver receiver;

  public CommandB(Receiver receiver) {
    this.receiver = receiver;
  }

  public void execute() {
    receiver.action2();
  }
}

public class Receiver {

  public void action1() {
    // implementation
  }

  public void action2() {
    // implementation
  }
}
```

A simple example how to apply the command pattern is given in Listing 27. Here, the instances of the concrete commands are created directly. Usually this will be done with a factory.

Listing 27: Application of the object oriented command pattern

```
public static void main(String[] args) {
```

```
  Receiver receiver1 = new Receiver();
  Receiver receiver2 = new Receiver();
  Invoker invoker = new Invoker();
  invoker.register(new CommandA(receiver1));
  invoker.execute();
  invoker.register(new CommandB(receiver2));
  invoker.execute();
  invoker.register(new CommandA(receiver2));
  invoker.execute();
}
```

5.2 An alternative approach using an enumeration

Just as with the patterns described earlier, it is possible to replace the inheritance relation with an enumeration, see Figure 7 and Listing 28. As the concrete commands now are enumeration values, it is not possible to store the receiver in the concrete commands [2]. Therefore, the receiver is stored with the invoker, and passed to the command when the command is to be executed.

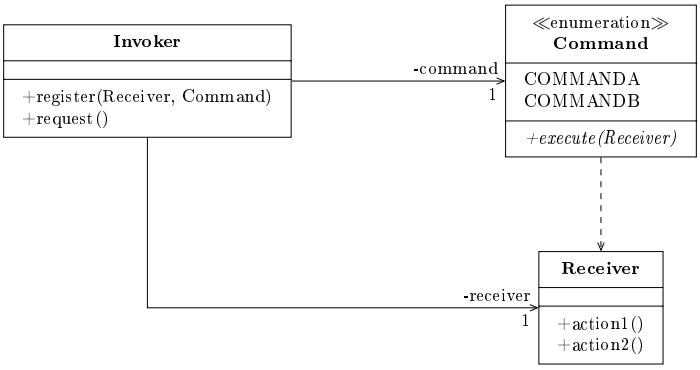

Figure 7: The command pattern, enumeration approach (Listing 28)

Listing 28: The command pattern, enumeration approach

```
public class Invoker {
  private Command command;
  private Receiver receiver;

  public void register(Receiver receiver, Command command) {
    this.receiver = receiver;
    this.command = command;
```

[2] Technically it is possible to add an attribute to an enumeration. However, as there is only one "instance" of the enumeration (constant), the value of the attribute is shared amongst all users of the enumeration (constant), which may cause unwanted effects

```
  }

  public void request() {
    command.execute(receiver);
  }
}

public enum Command {

  COMMAND1 {
    public void execute(Receiver receiver) {
      receiver.action1();

    }
  },
  COMMAND2 {
    public void execute(Receiver receiver) {
      receiver.action2();
    }
  };

  public abstract void execute(Receiver receiver);
}

public class Receiver {

  public void action1() {
    // implementation
  }

  public void action2() {
    // implementation
  }
}
```

A simple example how to apply the command pattern is given in Listing 29.

Listing 29: Application of the enumeration command pattern

```
public static void main(String[] args) {
  Receiver receiver1 = new Receiver();
  Receiver receiver2 = new Receiver();
  Invoker invoker = new Invoker();
  invoker.register(receiver1, Command.COMMAND1);
  invoker.execute();
  invoker.register(receiver2, Command.COMMAND2);
  invoker.execute();
  invoker.register(receiver2, Command.CommandA);
  invoker.execute();
}
```

A disadvantage of this approach is that the receiver cannot be hidden for the invoker. The invoker has to know the receiver, as the command cannot

store it. You can therefore argue whether this approach meets the intent of the commmand pattern.

5.3 An alternative approach using functional interfaces

With the use of functional interfaces, the command objects are not needed anymore. The receiver object with its required action can be registered directly with the invoker using the method reference "::". Figure 8 and Listing 30 show the command pattern in functional style.

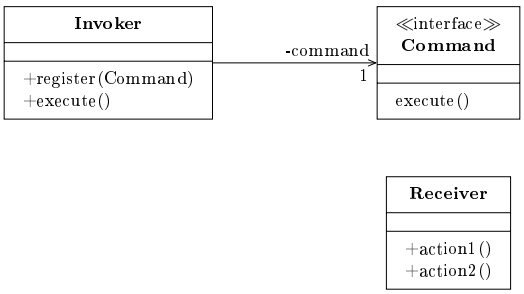

Figure 8: The command pattern, functional approach (Listing 30)

Listing 30: The command pattern, functional approach

```
public interface Command {
  void execute();
}

public class Invoker {
  private Command command;

  public void register(Command command) {
    this.command = command;
  }

  public void execute() {
    command.execute();
  }
}

public class Receiver {

  public void action1() {
    // implementation
  }

  public void action2() {
    // implementation
  }
}
```

In Listing 31 is shown how the receiver–action operation is registered with the invoker object in functional style. It is even possible to register commands using lambda expressions.

Listing 31: Application of the functional command pattern

```
public static void main(String[] args) {
  Receiver receiver1 = new Receiver();
  Receiver receiver2 = new Receiver();
  Invoker invoker = new Invoker();
  invoker.register(receiver1::action1);
  invoker.execute();
  invoker.register(receiver2::action2);
  invoker.execute();
  invoker.register(receiver2::action1);
  invoker.execute();
  invoker.register(()->System.out.println("extra command"));
  invoker.execute();
  invoker.register(new Receiver()->action2());
  invoker.execute();
}
```

5.4 Discussion of the approaches

The enumeration approach does not represent the command pattern properly. Of the other approaches the functional one clearly has the most simple structure and implementation. The Command objects are not needed anymore, as we register the receiver with it's action directly with the Invoker. However, when the concrete commands contain a complex structure (attributes), then the object oriented solution remains the best choice.

6 The Decorator pattern

The Decorator pattern can be applied when it is necessary to add functionality to an object at runtime without using sub classing.

The Decorator pattern is classified as a structural pattern in the GoF catalog. When the added functionality only consists of functions (algorithms), then the approaches we used for the other (behavioral) patterns can also be applied.

6.1 Applying the standard object oriented pattern

Figure 9 represents the basic object oriented pattern [9]. A concrete component of class `ComponentA` or `ComponentB` is decorated with one or more decorations (extra functionality in the form of method `addedBehavior`) defined by subclasses of the abstract class `Decorator`. The code is given in Listing 32.

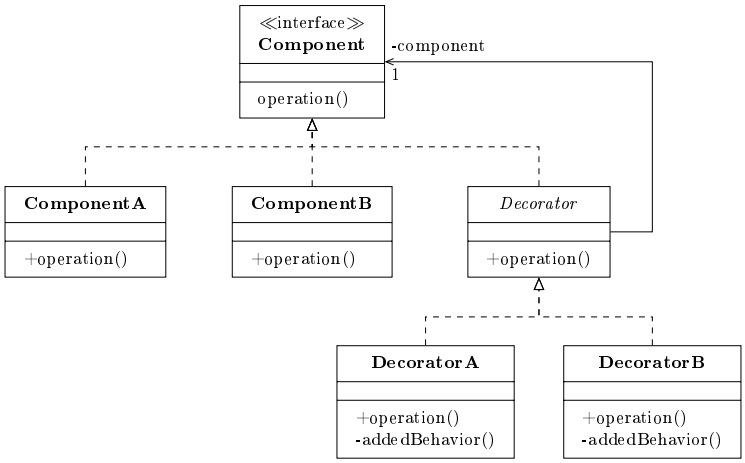

Figure 9: The decorator pattern, object oriented approach (Listing 32)

Listing 32: The strategy pattern, object oriented approach

```
public interface Component {
  void operation();
}

public class ComponentA implements Component {
  public void operation() {
    // implementation
  }
}

public class ComponentB implements Component {
  public void operation() {
```

```
    // implementation
  }
}

public abstract class Decorator implements Component {
  private Component component;

  protected Decorator(Component component) {
    this.component = component;
  }

  public void operation() {
    component.operation();
  }
}

public class DecoratorA extends Decorator {

  protected DecoratorA(Component component) {
    super(component);
  }

  public void operation() {
    super.operation();      // also possible in reverse order
    this.addedBehavior();   // so first addedBahavior
  }

  private void addedBehavior() {
    // implementation
  }
}

public class DecoratorB extends Decorator {

  protected DecoratorB(Component component) {
    super(component);
  }

  public void operation() {
    super.operation();
    this.addedBehavior();
  }

  private void addedBehavior() {
    // implementation
  }
}
```

A simple example how to apply the Decorator pattern is given in Listing 33. Concrete component A is decorated with three decorators, first with decorator A, and then twice with decorator B.

Here, the instances of the concrete components and concrete decorators are created directly. Usually this will be done with a factory.

Listing 33: Application of the object oriented decorator pattern

```
public static void main(String[] args) {
  Component comp = new DecoratorB(
                   new DecoratorB(
                   new DecoratorA(
                   new ComponentA())));
  comp.operation();
}
```

6.2 An alternative approach using an enumeration

Again, we can replace the class hierarchy by an enumeration, see Figure 10 and Listing 34. Enumeration `Decoration` contains a constant for each concrete decorator that contains an implementation of the method `addedBehavior`. Class `Decorator` is not abstract anymore. It stores the enumeration constant for the decoration it adds.

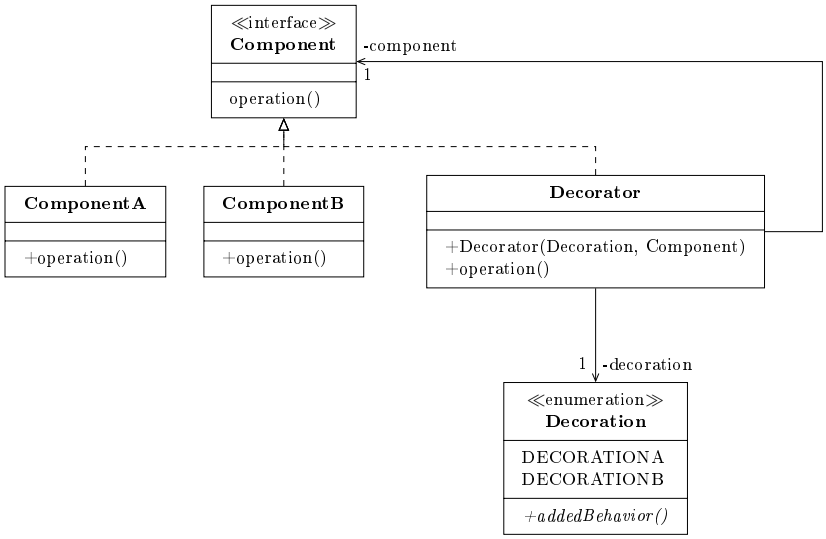

Figure 10: The Decorator pattern, enum approach (Listing 34)

Listing 34: The decorator pattern, enum approach

```
public interface Component {
  void operation();
}

public class ComponentA implements Component {
```

```
  public void operation() {
    // implementation
  }
}

public class ComponentB implements Component {
  public void operation() {
    // implementation
  }
}

public class Decorator implements Component {
  private Component component;
  private Decoration decoration;

  protected Decorator(Decoration decoration, Component component) {
    this.component = component;
    this.decoration = decoration;
  }

  public void operation() {
    component.operation();
    decorator.addedBehavior();
  }
}

public enum Decoration {

    DECORATIONA {
      public void addedBehavior() {
         // implementation
      }
    },

    DECORATIONB {
      public void addedBehavior() {
         // implementation
      }
    };

    public abstract void addedBehavior();
}
```

Listing 35 shows how to apply the enum-approach of the Decorator pattern to achieve the same results as in the object oriented approach of Listing 33.

Listing 35: Application of the enum oriented decorator pattern

```
public static void main(String[] args) {
  Component comp = new Decorator(Decoration.DECORATIONB,
                   new Decorator(Decoration.DECORATIONB,
                   new Decorator(Decoration.DECORATIONA,
                   new ComponentA())));
```

```
  comp.operation();
}
```

In this example we have to create a new decorator object for each decoration, only to add a new enumeration constant to the concrete component. We therefore also can chose to store all these decorations (enumeration constants) into a collection of one `Decorator` object. Using the Decorator pattern becomes easier, as shown in Listing 36.

Listing 36: Application of the enum oriented decorator pattern that uses a list

```
public static void main(String[] args) {
  Component comp = new Decorator(
                        new ComponentA(),
                        Decoration.DECORATIONB,
                        Decoration.DECORATIONB,
                        Decoration.DECORATIONA);
  comp.operation();
}
```

This solution requires modification of the class `Decorator`, see Figure 11 and Listing 37. Note that in this version it is still possible to add the decorations one at a time, as in Listing 35.

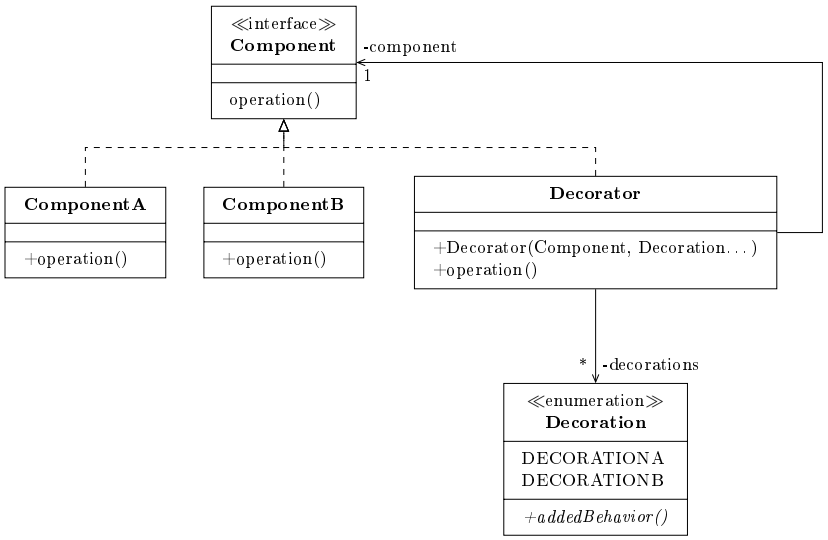

Figure 11: The Decorator pattern, list with enum approach (Listing 37)

Listing 37: The decorator pattern, modified Decorator class for list with enum approach

```
public class Decorator implements Component {
  private Component component;
  private Decoration[] decorations;

  protected Decorator(Component component,
                      Decoration ...decorations) {
    this.component = component;
    this.decorations = decorations;
  }

  public void operation() {
    component.operation();
    for (Decoration d : decorations) {
      d.addedBehavior();
    }
  }
}
```

6.3 Discussion of approaches

As with previous patterns the enumeration approach simplifies the class structure and implementation.

However, the object oriented approach is the most general solution. Not only pure functions can be used as decoration, but the decoration can also contain extra structure (i.e. attributes). An example of the Decorator pattern is the Java Stream API (for reading and writing). For example, abstract decorator `java.io.Reader` has a concrete decorator `java.io.BufferedReader` that adds several attributes to implement the buffer.

The enumeration approach is less applicable in situations where structure is added by a decoration, as they do not offer an easy way to store these attributes. When te decorations are pure function, for example in the famous coffee example of [4], then the enumeration and functional approach are both applicable, and simplify the implementation.

Another (small) disadvantage of the enumeration is that the signature of the method `addedBehavior` must be similar for all decorations. In the object oriented approach method `addedBehavior` can have another signature in each subclass, as the method is called from within the same subclass.

7 The Visitor pattern

The Visitor design pattern is used when similar operations are to be performed on the nodes of a class hierarchy, which we will call the Element classes. If this Element class hierarchy is relatively fixed but new operations need to be added easily, the advocated solution is to create a second class hierarchy, to be called the Visitor classes, with one node for every kind of operation. Each Visitor class will contain a separate implementation for every Element class. Now adding a new operation requires no more than adding a new Visitor class, while the Element classes remain unchanged, but adding a new Element class requires replacing all Visitor classes with versions having an extra implementation.

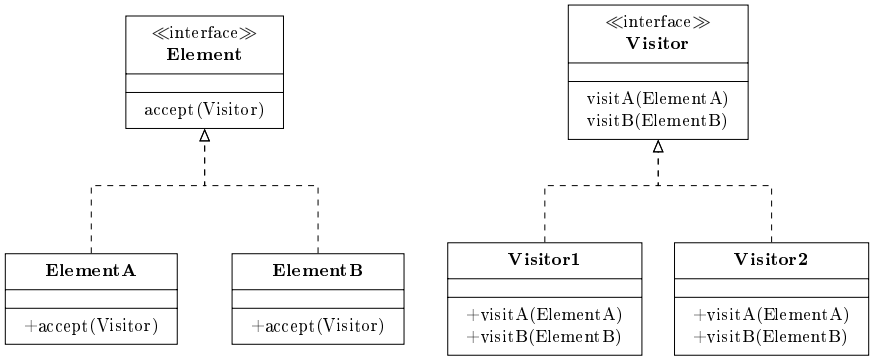

Figure 12: The Visitor pattern

Now insofar as the operations can be regarded as algorithms, it is conceivable to repeat the solution found for the Strategy pattern, letting all operations implement a functional interface. However, in this case it does not solve the problem the pattern is designed for. Even if one uses functional interfaces, the problem remains that the number of different implementations is quadratic (a separate one for each combination of Element and Visitor) and that these must be organized in some way without introducing a quadratic number of classes. Therefore introducing functional interfaces here does not essentially change the shape of the pattern; in fact, it complicates it because a separate functional interface is needed for every element type.

In the case of the Strategy pattern, our approach did not just simplify the code: it also provided for a better match with our mental model, because intuitively an algorithm is indeed a pure function rather than a class. With the Visitor pattern, this argument is less convincing: the operations are assumed to have a different implementation for each Element, hence will be thought of as methods in the Element classes rather than pure functions. This leads to a valid criticism of the original Visitor pattern too: the operation classes are inordinately interested in the workings of the Element classes, hence suffer from the code smell *Feature Envy* [3]. And because Java lacks a mechanism like C++ friends, this requires the Element classes to provide public access to all features

necessary to perform the operations.

8 UML extension: a proposal

In Section 3 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 one of the functional interfaces listed in Table 1. 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 13. 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.

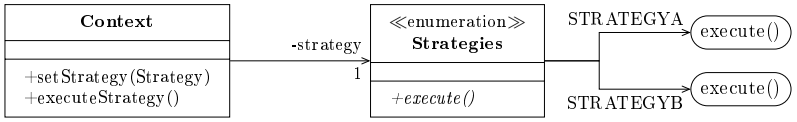

Figure 13: The strategy pattern, enum style (Listing 17)

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.

9 Related work

It has been observed many times before that design patterns reflect a lack of features in programming languages. The GoF patterns [9] 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 [18] showed that using a more permissive object-oriented language would make some design patterns disappear. Hannemann and Kiczales [10] 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 [5, 6, 7, 8]. 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 [11] 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.

10 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.

10.1 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 [12]. 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 [17], 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. For instance, the interface `Strategy` in Listing 19 is only necessary because lambda expressions in Java do not have a inbuilt type allowing a call to `execute()`. 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] Martin Fowler, Steven Fraser, Kent Beck, Bil Caputo, Tim Mackinnon, James Newkirk, and Charlie Poole. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

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

[15] 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.

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

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

[18] 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.

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

[20] Richard Warburton. *Java 8 Lambdas: Pragmatic Functional Programming.* O'Reilly Media, Inc., 1 edition, 2014.

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