The Strategy Configuration Problem and How to Solve It

Apostolos V. Zarras
Department of Computer Science and Engineering, University of Ioannina
Greece
zarras@cs.uoi.gr

ABSTRACT

The Strategy pattern allows the developer to implement a family of algorithms that can be used interchangeably and vary independently from the objects that use them. To achieve this, the algorithms are implemented as an hierarchy of respective strategy classes that realize the same interface.

One particular issue that is not precisely specified in the pattern is how to configure objects with the algorithms that they need to use. This paper introduces recurring solutions to the problem. These solutions appeared in different projects delivered in the context of Software Engineering, a compulsory course of the Department of Computer Science and Engineering of the University of Ioannina. The reported solutions are of two kinds, those that facilitate the *constant configuration* of objects with algorithms that do not change during the lifetime of the objects, and those that enable the *adaptable configuration* of objects with algorithms that can be dynamically reconfigured. The solutions that adhere to the intent of Strategy are reported as *patterns*, while the solutions that deviate from it are reported as *anti-patterns*.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**;

KEYWORDS

Behavioral Patterns, Strategy

ACM Reference Format:
Apostolos V. Zarras. 2021. The Strategy Configuration Problem and How to Solve It. In *European Conference on Pattern Languages of Programs (EuroPLoP'21), July 7–11, 2021, Graz, Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3489449.3489980

1 INTRODUCTION

The Gang of Four (GoF) patterns [4] provide reusable solutions to problems that are frequently encountered in the development of Object-Oriented (OO) software. The usefulness of the patterns has been proven over the years by the fact that they are diachronic, still being used by many developers in many different contexts and OO programming languages. Apart from their practical application, over the years there have also been several research papers, investigating the way that they are applied, their benefits and liabilities (e.g., [1–3, 5, 8–10]).

The key issues for getting the best out of the GoF patterns (and in fact out of all patterns) is to select the right pattern for the problem in hand, and to implement it correctly in this particular context. However, this is not always the case. In practice, the misuse or the incorrect implementation of patterns can introduce problems, increase the complexity and deteriorate the overall software quality [11]. For this reason, several approaches emerged for checking the correct use of design patterns [6].

Typically, the mistakes that people do when they apply a particular pattern are recurring [12]. Trying to find common errors and misuses in the use of design patterns and documenting them, along with solutions that allow the developers to avoid them is an interesting and challenging exercise. Since the observed errors and misuses are recurring they qualify as anti-patterns, i.e., ineffective solutions to frequently encountered problems that introduce risks, inefficiencies and other problems [7]. A number of such anti-patterns that concern mistakes in the use of the GoF Command pattern are discussed in [12]. Moreover, [12] reports a pattern that allows to avoid these mistakes.

This paper, focuses on the Strategy pattern. Strategy is a very handy behavioral pattern, found in the GoF catalog [4]. The intent of the pattern is to let a developer implement *a family of alternative algorithms* in a way that allows them to be used interchangeably and vary independently from the objects that use them. To this end, the algorithms should be implemented as *an hierarchy of concrete strategy classes* that implement *a common interface.*

> ***Although the description of the pattern in the GoF catalog is clear and simple, when it comes to the actual use of the pattern, a key point that demands special attention is that there are several approaches that can be followed to configure an object with the algorithm it will use. The situations in which these approaches can be applied and the consequences of the approaches vary.***

This paper, discusses different solutions to the aforementioned problem. These solutions appeared in projects delivered in the context of Software Engineering, a course that took place in the second semester of 2019-2020 at the Department of Computer Science and Engineering of the University of Ioannina. Software Engineering is a compulsory course in the Department's five-year curriculum. The course is offered to fourth-year students of the Department. During the course the students study conventional and agile software development methods. A core part of the course is the study of popular design patterns from the GoF catalog and the use of these patterns in the project of the course.

EuroPLoP'21, July 7–11, 2021, Graz, Austria

ACM ISBN 978-1-4503-8997-6/21/07…$15.00
https://doi.org/10.1145/3489449.3489980

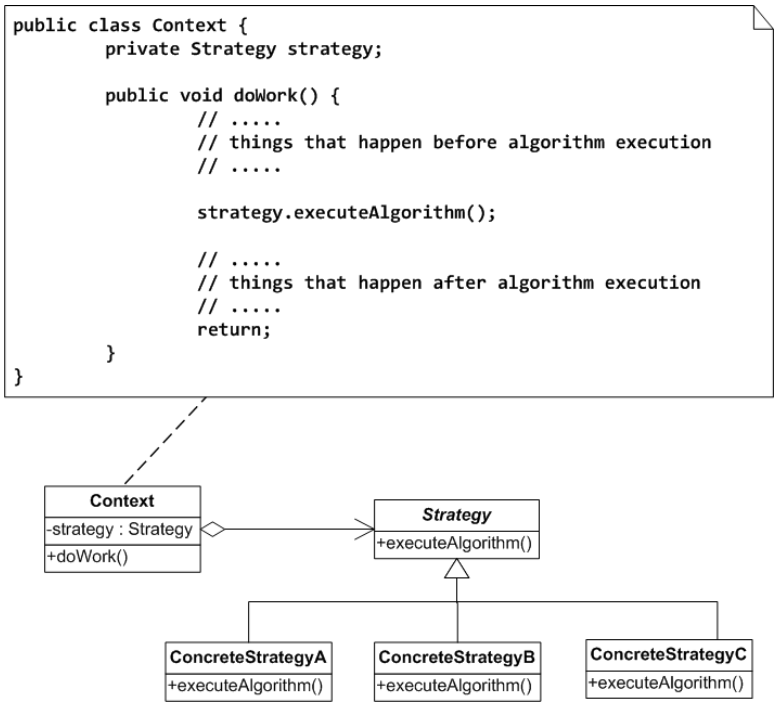

Figure 1: The structure of the Strategy pattern.

The observed solutions are divided in two categories: *The first category includes solutions that facilitate the constant configuration of objects with algorithms that do not change during the lifetime of the objects, while the second category comprises solutions that enable the adaptable configuration of objects with algorithms that can by dynamically reconfigured.* The solutions of each category are further divided into *anti-patterns* that compromise the intent of Strategy and *patterns* that conform with Strategy. The paper further provides a detailed *post-mortem* analysis of the use of the anti-patterns/patterns in the projects of the Software Engineering course.

The rest of this paper is structured as follows. Section 2, discusses in more detail the structure and the benefits of the Strategy pattern. Sections 3 and 4 detail the constant and the adaptable configuration anti-patterns/patterns, respectively. Section 5, discusses the anti-patterns/patterns mining process and the uses of the reported anti-patterns/patterns, in the context of the Software Engineering course. Finally, Section 6 summarizes the contribution of this paper.

2 STRATEGY PATTERN

Intent

Literally, the intent of the Strategy pattern is the following [4]:

> ***"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."***

Simply put, there are three key ideas behind the pattern:

- The first idea is to encapsulate alternative algorithms, so that the objects that use them are not exposed to the internal implementation details of the algorithms. To achieve the encapsulation, the algorithms are implemented in separate classes.
- The second idea is to make the encapsulated algorithms interchangeable so that any one can be used in place of the other. To make the algorithms interchangeable all the classes that realize the algorithms implement a common interface.
- The third idea is to let the algorithms vary independently from the objects that use them. To this end, the objects use the common interface to call the algorithms.

Structure

In more detail, the structure of the Strategy pattern involves the following participants (Figure 1):

- `Strategy`, defines a common interface for all the algorithms that belong to the family.
- `ConcreteStrategyA`, `ConcreteStrategyB`, ... are different implementations of `Strategy`, one for each alternative algorithm that belongs to the family.
- `Context`, uses an algorithm to do some work. To this end, `Context` has a reference to a `Strategy` object.

In a typical execution scenario the pattern participants collaborate as follows: A client configures a `Context` object with an algorithm by setting the `Strategy` reference to a corresponding object of a concrete class that implements `Strategy`. The `Context` object interacts with the referenced object by invoking methods

of the Strategy interface. In general, there are several ways for configuring the Context object with an algorithm. Some of these ways, discussed in Section 3, result in a constant configuration that can not be changed during the lifetime of the Context object, while others, detailed in Section 4, facilitate the adaptable configuration of the Context object.

3 CONSTANT STRATEGY CONFIGURATION

Context

A developer implements a family of algorithms. The objects of a Context class have to use these algorithms. The developer employs Strategy to encapsulate the algorithms, make them interchangeable, and let them vary independently from the Context objects that use them. The algorithm that is used by a Context object does not have to be changed with another one during the lifetime of the object.

Problem

The developer should implement the code that configures Context objects with algorithms and *make sure that the resulting configurations are constant*. The design choices that he will make to achieve this goal should not deviate from the intent of the Strategy pattern.

Forces

- The configuration code must ensure that the algorithms used by the Context objects are specified once and do not change with other algorithms from then on.
- The configuration code should facilitate the interchangeability of the algorithms.
- The configuration code should contribute to the independent variation of the algorithms from the Context objects that use them.

Anti-Pattern: Strategy Creation in Constructor

The developer encapsulates the algorithms in separate concrete classes that implement the Strategy interface (Figure 2). The developer further defines a Strategy field in Context and a Context constructor, which takes a parameter, whose value identifies a concrete class that implements Strategy; hereafter the term *strategy identifier* is used to refer to such parameter. The Context constructor is the only means for setting the value of strategy. Specifically, the configuration of a Context object with an algorithm, takes place when the object is created by a Client object. The Client object gives to the constructor a strategy identifier. Based on the given strategy identifier, the constructor selects the algorithm, creates an object of the respective concrete class, and assigns to strategy a reference to this object.

Consequences.

+ The configuration of a Context object is done when the object is created and can not be changed after this point.
+ A new Context object can be configured with any algorithm, by giving to the constructor a corresponding strategy identifier.
- The parameterized constructor depends on the concrete classes that realize the algorithms. Hence, any addition, deletion or modification to the family of algorithms can affect Context.
- The use of a strategy identifier as parameter may not be very intuitive and type safe.
- The parameterized constructor includes a complex selection logic for choosing the particular algorithm to use.
- Giving an invalid strategy identifier to the parameterized constructor can result to a Context object that does not behave properly. Configuring the Context object with a default algorithm can be a way to avoid the problem. However, it may not always be possible to consider an algorithm as a default.

Pattern: Strategy Injection with Constructor

The developer implements the algorithms as separate concrete classes that realize the Strategy interface (Figure 3). Moreover, the developer defines a Strategy field in Context and a Context constructor, which takes as parameter a Strategy reference. The parameterized constructor is the only means for setting the value of strategy. In a typical execution scenario, a Client object creates a Context object. To configure the Context object with an algorithm, the Client object passes to the constructor a reference to an object of the corresponding concrete class. The constructor assigns the given object reference to strategy.

Consequences.

+ The configuration of a Context object with an algorithm is done once and for all by the parameterized constructor, during the creation of the Context object.
+ A new Context object can be configured with any algorithm, by passing to the constructor an reference to an object of the concrete class that implements the algorithm.
+ The parameterized constructor is not coupled with the concrete classes that realize the algorithms. Hence, any addition, deletion or modifications to the family of algorithms can be done without changing Context.
- Passing null to the parameterized constructor can result in a Context object that does not behave correctly. The issue can be handled by configuring the Context object with a default algorithm. However, it may not always be possible to assume a default algorithm.

Variants

There are certain variants of the pattern solution that the developer can use to deal with the constant strategy configuration problem. The developer's choice should be done on the basis of the consequences that characterize the basic solution and its variants.

Constant Strategy Configuration Using Reflection. The developer defines a Context constructor that takes as parameter a strategy identifier. The constructor uses reflection[1] to create a new object that belongs to the corresponding concrete class and assigns to strategy a reference to this object (Figure 4). Reflection allows

[1] en.wikipedia.org/wiki/Reflective_programming#cite_note-1

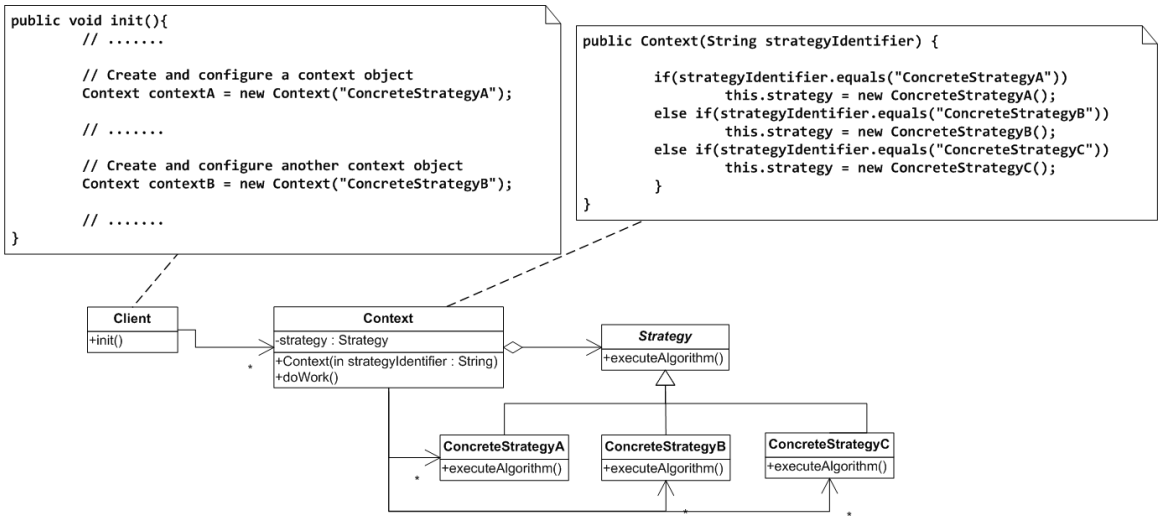

Figure 2: Constant Strategy Configuration Anti-Pattern: Strategy Creation in Constructor.

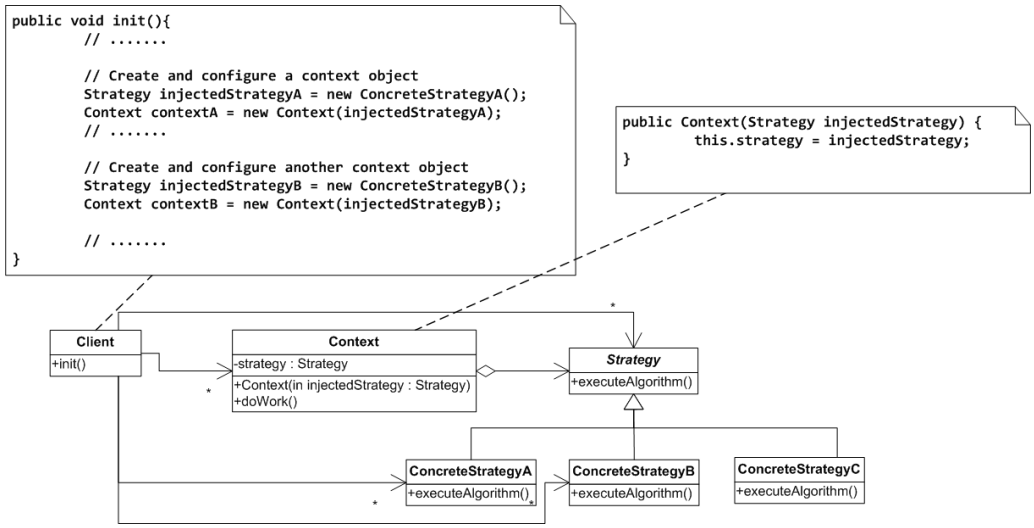

Figure 3: Constant Strategy Configuration Pattern: Strategy Injection with Constructor.

to create the object, without explicitly referring to the class of the object. To determine the class of the object, it is sufficient to pass to the reflection mechanism the given strategy identifier.

Consequences.

+ Reflection decouples `Context` from the concrete classes that implement `Strategy`.
+ `Client` is not coupled with the concrete classes that implement `Strategy`.
- Decoupling `Context` from the concrete classes may not be possible if the constructors of the concrete classes require different parameters.
- Reflection can make the parameterized constructor more complicated and harder to understand.
- The use of reflection may introduce an additional performance overhead.
- Giving an invalid strategy identifier to the parameterized constructor can result to a `Context` object that does not behave correctly.

Constant Strategy Configuration With Parameterized Factory. The developer defines a `Context` constructor that takes as parameter a strategy identifier. Moreover, the developer implements a parameterized factory for the concrete classes that implement `Strategy` [4] (Figure 5). The parameterized factory provides a factory method that takes as parameter a strategy identifier and returns a reference to a `Strategy` object. The `Context` constructor gives the strategy identifier to the factory method. The factory method

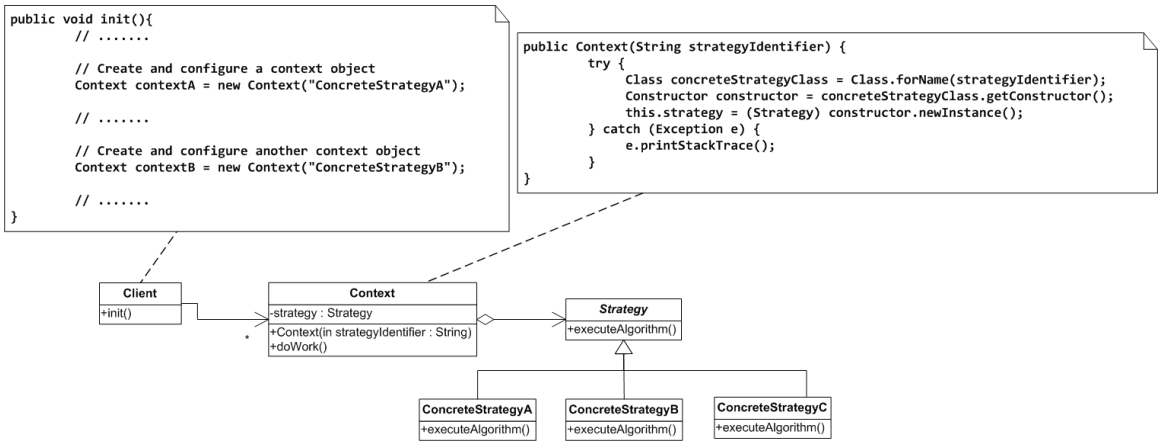

Figure 4: Constant Strategy Configuration Using Reflection.

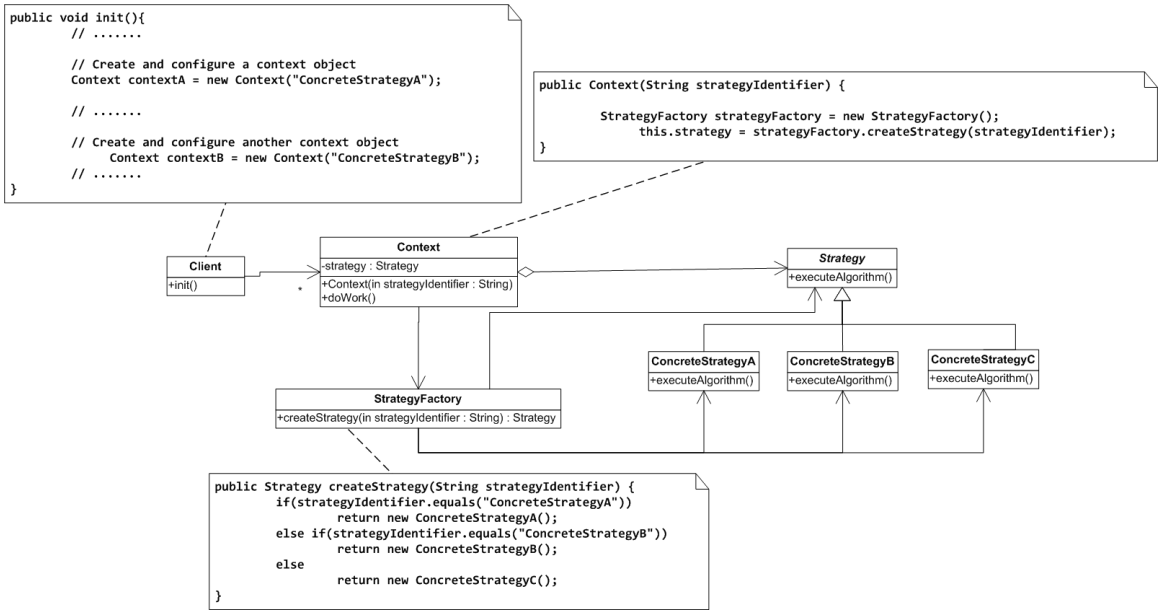

Figure 5: Constant Strategy Configuration With Parameterized Factory.

creates an object of the corresponding concrete class and returns a reference to the object. Finally, the constructor assigns the returned reference to `strategy`.

Consequences.

+ The parameterized factory allows to decouple `Context` from the concrete classes that implement `Strategy`.
+ `Client` is not coupled with the concrete classes that implement `Strategy`.
- The parameterized factory is an extra class that is added in the overall design.
- The parameterized factory is coupled with the concrete classes that implement `Strategy`.
- Giving an invalid strategy identifier to the parameterized factory can result to a `Context` object that does not behave correctly.

4 ADAPTABLE STRATEGY CONFIGURATION

Context

A developer implements a family of algorithms. The objects of a `Context` class have to use these algorithms. The developer employs STRATEGY to encapsulate the algorithms, make them interchangeable, and let them vary independently from the `Context` objects that use them. The algorithm needed by a `Context` object *may be specified at any time, during the lifetime of the object.* Moreover, the

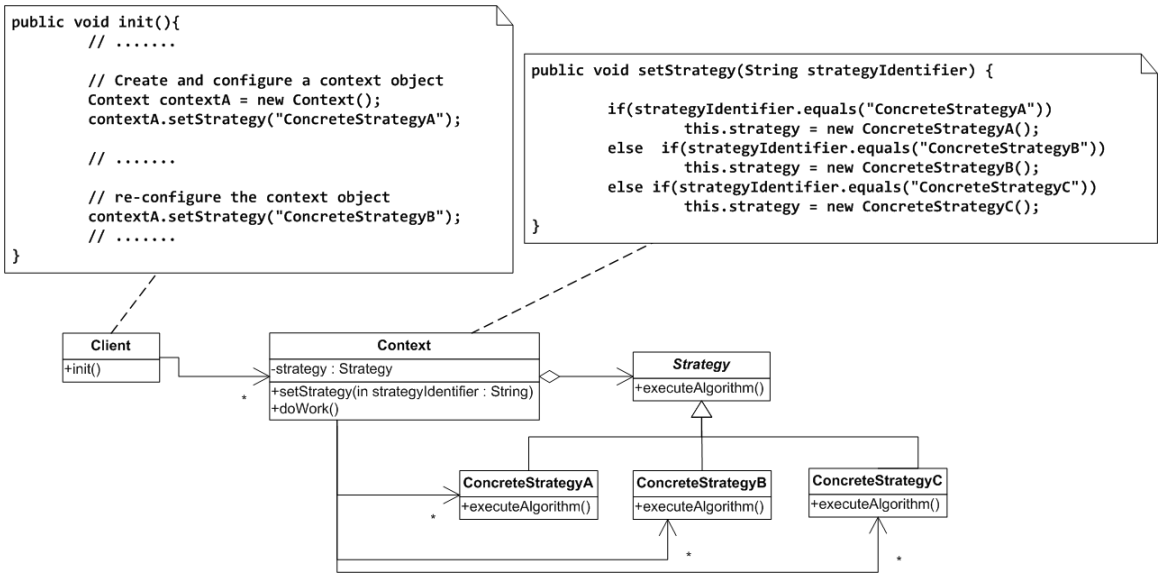

Figure 6: Adaptable Strategy Configuration Anti-Pattern: Strategy Creation in Setter.

algorithm that is used by the object *may be replaced with another algorithm, during the lifetime of the object.*

Problem

The developer should implement the code that configures `Context` objects with algorithms and *make sure that the resulting configurations are adaptable.* The developer's design choices towards the implementation of the configuration code should comply with the intent of the Strategy pattern.

Forces

- The configuration code should allow to configure and alter the algorithms used by the `Context` objects at any time.
- The configuration code should enable the interchangeability of the algorithms.
- The configuration code should facilitate the independent variation of the algorithms from the `Context` objects that use them.

Anti-Pattern: Strategy Creation in Setter

The developer encapsulates the algorithms in separate concrete classes that implement the `Strategy` interface (Figure 6). The developer further defines a `Strategy` field in `Context` and a method for setting the value of this field. The setter method takes as parameter a strategy identifier. To configure `Context` object with an algorithm, a `Client` object calls the setter method on the `Context` object, with a strategy identifier as parameter. The setter method creates an object of the corresponding concrete class and assigns to `strategy` a reference to this object. To re-configure the `Context` object with another algorithm, the `Client` object should re-invoke the setter method on the `Context` object, with the strategy identifier of the concrete class that realizes this other algorithm as parameter.

Consequences.

+ The configuration of a `Context` object can be changed at any time by re-invoking the setter method.
+ A `Context` object can be configured with any algorithm, by giving to the setter method a corresponding strategy identifier.
- The setter method is coupled with the concrete classes that realize the algorithms. Therefore, any addition, deletion or modification to the family of algorithms can affect `Context`.
- The use of a strategy identifier as parameter may not be very intuitive and type safe.
- The setter method comprises a complex selection logic for selecting the particular algorithm to use.
- Passing an invalid strategy identifier when calling the setter method on a `Context` object can cause erroneous object behaviors. Configuring the `Context` object with a default algorithm may be a possible solution to this issue, assuming that it is possible to consider an algorithm as a default.

Pattern: Strategy Injection with Setter

The developer implements the algorithms as separate concrete classes that realize the `Strategy` interface (Figure 7). The developer further defines a `Strategy` field in `Context` and a method for setting the value of this field. The setter method takes as parameter a reference to a `Strategy` object. To configure a `Context` object with an algorithm, a `Client` object calls the setter method on the `Context` object, with a reference to an object of the corresponding concrete class as parameter. The setter method assigns to `strategy` the given object reference. To re-configure the `Context` object with another algorithm, the `Client` object should re-invoke the setter method on the `Context` object, with a reference to an object of the concrete class that realizes this other algorithm as parameter.

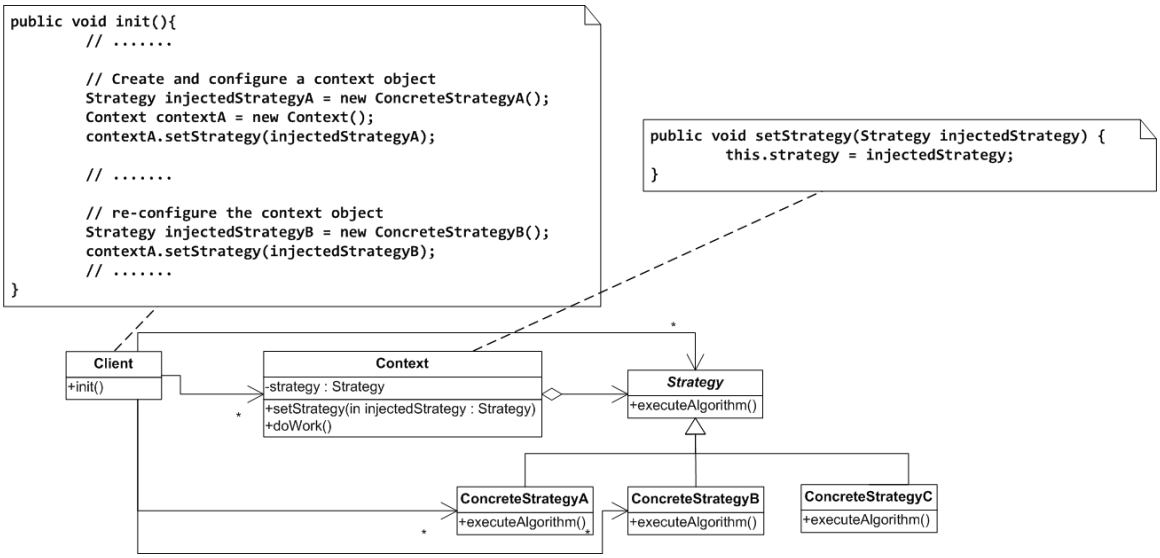

Figure 7: Adaptable Strategy Configuration Pattern: Strategy Injection with Setter.

Consequences.

+ The configuration of a `Context` object can be altered at any time by re-invoking the setter method.
+ A `Context` object can be configured with any algorithm, by giving to the setter method a reference to an object of the corresponding concrete class.
+ The setter method does not depend on the concrete classes that realize the algorithms. Thus, any addition, deletion or modification to the family of algorithms does not require to change `Context`.
- Passing null when calling the setter method on a `Context` object can cause erroneous object behaviors. Configuring the `Context` object with a default algorithm may be a possible solution to overcome this issue. Nevertheless, it may not always be possible to consider an algorithm as a default.

Variants

There are certain variants of the pattern solution that the developer can use to deal with the adaptable strategy configuration problem. The developer's choice should be done with respect to the consequences that characterize the basic solution and its variants.

Adaptable Strategy Configuration Using Reflection. The developer defines a setter method in `Context` that takes as parameter a strategy identifier. The setter method uses reflection to create a new object that belongs to the concrete class and assigns to `strategy` a reference to this object (Figure 8).

Consequences.

+ Reflection allows to decouple `Context` from the concrete classes that implement `Strategy`.
+ `Client` is not coupled with the concrete classes that implement `Strategy`.
- Decoupling `Context` from the concrete classes may not be possible if the constructors of the concrete classes require different parameters.
- Reflection can make the setter method more complicated and harder to understand.
- The use of reflection may introduce an additional performance overhead.
- Passing an invalid strategy identifier when calling the setter method on a `Context` object can cause erroneous object behaviors.

Adaptable Strategy Configuration With Parameterized Factory. The developer defines a setter method in `Context` that takes as parameter a strategy identifier. The developer further implements a parameterized factory for the concrete classes that implement `Strategy` [4]. The parameterized factory provides a factory method that takes as parameter a strategy identifier and returns a reference to a `Strategy` object (Figure 9). The setter method gives the strategy identifier as input to the factory method. The factory method creates an object of the corresponding concrete class and returns a reference to the object. Finally, the setter method assigns the object reference to `strategy`.

Consequences.

+ The parameterized factory allows to decouple `Context` from the concrete classes that implement `Strategy`.
+ `Client` is not coupled with the concrete classes that implement `Strategy`.
- The parameterized factory is an extra class that is added in the overall design.
- The parameterized factory depends on the concrete classes that implement `Strategy`.
- Passing an invalid strategy identifier when calling the setter method on a `Context` object can cause erroneous object behaviors.

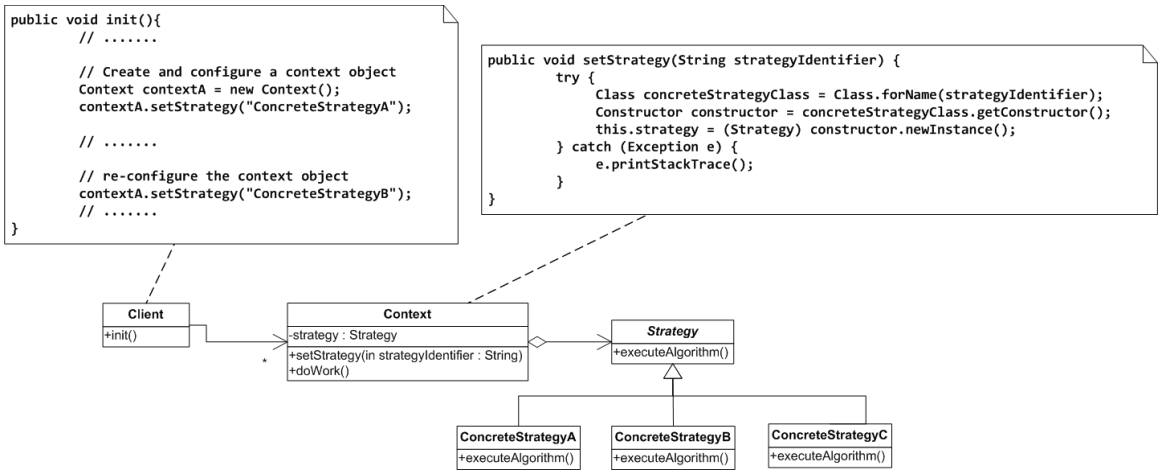

Figure 8: Adaptable Strategy Configuration Using Reflection.

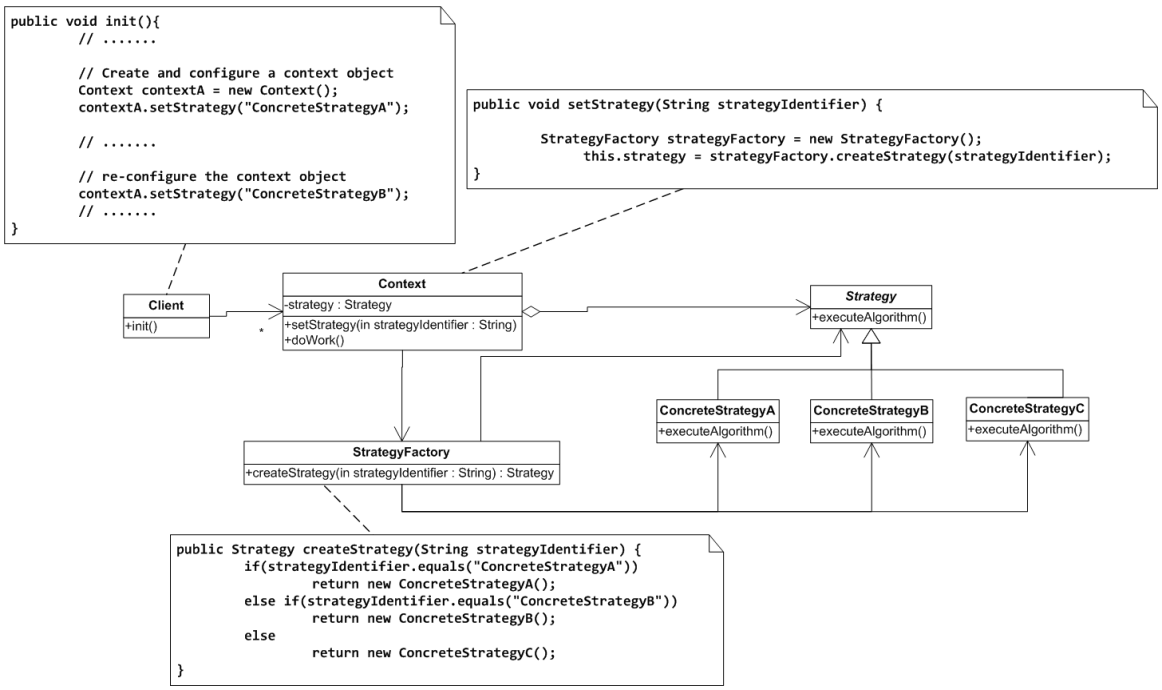

Figure 9: Adaptable Strategy Configuration With Parameterized Factory.

Adaptable Strategy Configuration With Map. In general, a map (or dictionary) is a generic data structure for storing object references. A map has a set of keys and each key has a single associated object reference. A main advantage of the map over a simple list or an array of object references is that the code that uses the map to retrieve an object reference that is associated with a particular key is quite simple. Typically, the map provides a method that takes as input a key and returns the associated object reference. The complexity of the algorithm that actually matches a given key against the keys stored in the map is encapsulated in this method.

In this variant, the idea is to associate strategy identifiers with object references of the corresponding concrete classes that implement `Strategy` by adding a map field in `Context` (Figure 10). `Context` further provides a setter method that takes as parameter a strategy identifier. The setter method uses the strategy identifier to get the associated object reference from the map. Finally, the setter method assigns the object reference to `strategy`.

One option for the initialization of the map field is to add a corresponding parameter in the `Context` constructor. Another option, is to initialize the map using reflection. In any case, the initialization of

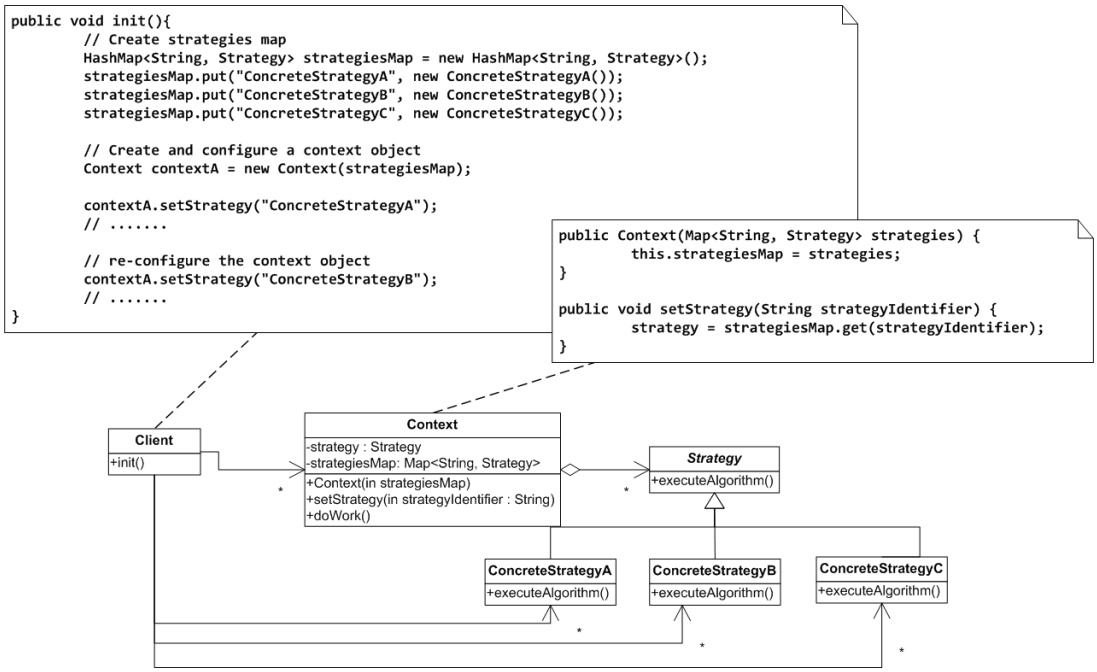

Figure 10: Adaptable Strategy Configuration with Map.

the map should be done in a way that keeps `Context` independent from the concrete classes that implement `Strategy`.

Consequences.

+ The map decouples `Context` from the concrete classes that implement `Strategy`.
- `Context` becomes more complex with the definition of the additional map field and the respective parameterized constructor.
- The use of the map can make the code harder to comprehend and introduce type safety issues.

Adaptable Strategy Configuration on Demand. The developer adds a `Strategy` reference as an extra parameter to all the methods of `Context` that need to use an algorithm. A `Client` object that calls a method on a `Context` object uses the extra parameter to pass a reference to an object of a concrete class that realizes an algorithm (Figure 11). The method uses this reference to execute the algorithm.

Consequences.

+ `Context` is decoupled from the concrete classes that implement `Strategy`.
- The prototypes of the methods that need to use an algorithm become more complex.
- Passing null when calling a method on a `Context` object can cause erroneous object behaviors.

5 EMPIRICAL EVIDENCE

As already mentioned in Section 1 the patterns and the anti-patterns reported in this paper were observed in projects delivered in the context of the Software Engineering course that took place in the second semester of 2019-2020 at the Department of Computer Science and Engineering of the University of Ioannina. The students of the course formed 58 different development groups consisting of 2-3 people. Each group developed its own project. The overall duration of the project was 10 weeks. The main objective of the project was to develop a simple text-to-speech editor. One of the requirements for the editor was to encode a given text and transform the encoded text to speech. Another requirement was to provide different kinds of encodings and allow the users to alter the particular encoding strategy that is used for the text transformation at any time. Following the instructor's guidance, the groups used the Strategy pattern to implement the encoding algorithms.

Concerning the configuration of context objects with algorithms, the groups employed different solutions. Some of these solutions conformed with the observed anti-patterns, while some others conformed with the observed patterns. According to the requirements of the project, the most appropriate pattern for the project would be the Strategy Injection with Setter. Figure 12, summarizes what the groups actually did in practice. Specifically, 44 groups groups successfully used the Strategy Injection with Setter (and its variants). Overall that is 75.86% of the groups that enrolled in the software engineering course. The remaining groups did not make the right design choices; either they used an anti-pattern or a pattern that is not appropriate for the context of the project. More specifically, 5 groups used Strategy Creation in Setter, 7 used the Strategy Injection with Constructor, and 2 used the Strategy Creation in Constructor.

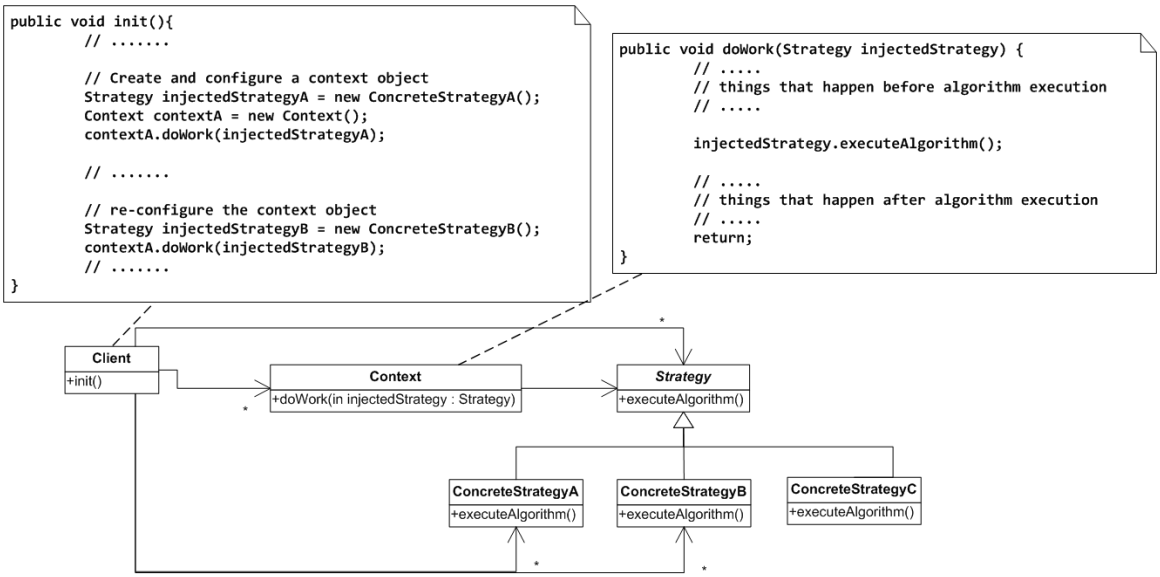

Figure 11: Adaptable Strategy Configuration on Demand.

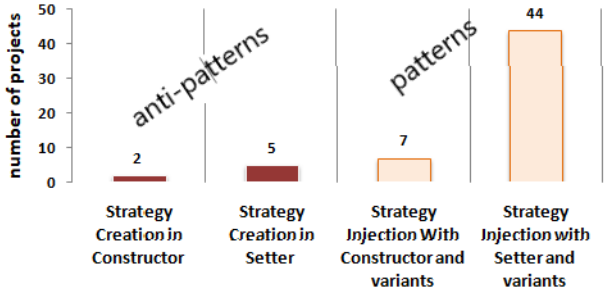

Figure 12: Anti-patterns/patterns uses in the software engineering course project.

6 CONCLUSION

Developers often make mistakes when they try to apply a pattern in their application. Discovering and systematically documenting these mistakes can be very helpful for the developers. In this vein, this paper focused on the Strategy pattern. The paper introduced different anti-patterns and patterns that document, respectively, bad and good solutions to the problem of configuring context objects with strategies. The anti-patterns/patterns appeared in different projects delivered in the context of the Software Engineering course of the Department of Computer Science and Engineering of the University of Ioannina. The anti-patterns/patterns are of two kinds, those that facilitate the constant configuration of objects with algorithms that do not change during the lifetime of the objects, and those that enable the adaptable configuration of objects with algorithms that can be dynamically reconfigured. The impact of the observed anti-patterns/patterns on the degree to which the benefits of the Strategy pattern can be achieved was also discussed in detail.

ACKNOWLEDGMENTS

I would like to thank the shepherd of the paper, Eduardo Guerra, for his valuable comments and suggestions. I would also like to thank the members of the writers' workshop for their feedback on the paper.

REFERENCES

[1] Apostolos Ampatzoglou and Alexander Chatzigeorgiou. 2007. Evaluation of Object-Oriented Design Patterns in Game Development. *Information and Software Technology* 49, 5 (2007), 445–454.
[2] Lerina Aversano, Gerardo Canfora, Luigi Cerulo, Concettina Del Grosso, and Massimiliano Di Penta. 2007. An Empirical Study on the Evolution of Design Patterns. In *Proceedings of the 6th Joint European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE)*. 385–394.
[3] James M. Bieman, Greg Straw, Huxia Wang, Willard P. Munger, and Roger T. Alexander. 2003. Design Patterns and Change Proneness: An Examination of Five Evolving Systems. In *Proceedings of the 9th IEEE International Symposium on Software Metrics (METRICS)*. IEEE Computer Society, Washington, DC, USA, 40–50.
[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley.
[5] Brian Huston. 2001. The Effects of Design Pattern Application on Metric Scores. *Journal of Systems and Software* 58, 3 (2001), 261–269.
[6] Salman Khwaja and Mohammad Alshayeb. 2016. Survey On Software Design-Pattern Specification Languages. *ACM Compututing Surveys* 49, 1 (2016), 21:1–21:35.
[7] Andrew Koenig. 1995. Patterns and Antipatterns. *Journal of Object Oriented Programming (JOOP)* 8, 1 (1995), 46–48.
[8] L. Prechelt, B. Unger, W.F. Tichy, P. Brossler, and L.G. Votta. 2001. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. *IEEE Transactions on Software Engineering* 27, 12 (2001), 1134–1144.
[9] Lutz Prechelt, Barbara Unger-Lamprecht, Michael Philippsen, and Walter F. Tichy. 2002. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Transactions on Software Engineering* 28, 6 (2002), 595–606.
[10] Marek Vokac. 2004. Defect Frequency and Design Patterns: An Empirical Study of Industrial Code. *IEEE Transactions on Software Engineering* 30, 12 (2004), 904–917.

[11] Peter Wendorff. 2001. Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project. In *Proceedings of the IEEE European Conference on Software Maintenance and Reengineering (CSMR)*. 77–87.

[12] Apostolos V. Zarras. 2020. Common Mistakes When Using the Command Pattern and How to Avoid Them. In *Proceedings of the ACM European Conference on Pattern Languages of Programs (EuroPLoP)*. 4:1–4:9.