UNIVERSITY OF MACEDONIA
APPLIED INFORMATICS

SmartCLIDE Design Pattern Assistant:

A Decision-Tree based Approach

EVANGELIA PAPAGIANNAKI DAI19008
ELENI POLYZOIDOU DAI19188

BACHELOR THESIS

THESSALONIKI 2022

Table of Contents

1. Introduction 5
2. Literature Review 7
2.1 Design Patterns 7
2.1.1 Factory Method 7
2.1.2 Builder 8
2.1.3 Bridge 8
2.1.4 Composite 9
2.1.5 Command 9
2.1.6 Memento 10
2.1.7 Strategy 11
2.1.8 Observer 11
2.2 Pattern Selection Approaches 12
2.3 Benefits of Using Patterns 16
3. Proposed Solution 19
3.1 Pattern Selection Decision Trees 19
3.1.1 Creational Patterns Selection Decision Tree 19
3.1.2 Structural Patterns Selection Decision Tree 20
3.1.3 Behavioral Patterns Selection Decision Tree 21
3.2 Tool Support: Eclipse Theia Extension 27
3.2.1 Eclipse Theia 27
3.2.2 Presentation of the tool 27
3.2.2.1 Tool Walk-through 27
3.2.2.2 Pattern examples 33
3.2.3 Tool Architecture 44
4. Industrial Validation 57
4.1 Objectives & Research Questions 57
4.2 Industrial Study Setup 58
4.3 Data Collection & Analysis 59
5.Results 64
5.1 State-of-Practice and Expected Advancements 65
5.2 Correctness, Timeliness, and Usefulness of the SmartCLIDE Pattern Selection Approach 66
5.3 Usability Evaluation 68
6. Discussion 70
7. Conclusions 71
Reference 72

List of Tables

Table 1. Study Demographics 58
Table 2. Participants Assignment to Tasks 58
Table 3. Data Collection Methods per Research Question 59
Table 4. Codes of the Qualitative Analysis 64

List of Figures

Figure 1. Selection Decision Tree for Creational Design Patterns..... 24
Figure 2. Selection Decision Tree for Structural Design Patterns..... 25
Figure 3. Selection Decision Tree for Behavioral Design Patterns..... 26
Figure 4. Launching the Theia Extension..... 28
Figure 5. Welcome Screen..... 28
Figure 6. Pattern Selection in EXPERT-MODE..... 29
Figure 7. Pattern Selection in WIZARD-MODE..... 30
Figure 8. Autocompleter operation..... 31
Figure 9. Autocomplete operation..... 31
Figure 10. Plus button operation..... 32
Figure 11. Error message..... 33
Figure 12. Code Generation..... 33
Figure 13. Features Usefulness..... 68
Figure 14. Usability Evaluation Outcome..... 70

Abstract

Design patterns are well-known solutions to recurring design problems that are widely adopted in the software industry, either as formal means of communication or as a way to improve structural quality, enabling proper software extension. However, the adoption and correct instantiation of patterns is not a trivial task and requires substantial design experience. Some patterns are conceptually close or present similar design alternatives, leading novice developers to improper pattern selection, thereby reducing maintainability. Additionally, the mis-instantiation of a GoF design pattern, leads to phenomena such as pattern grime or architecture decay. To alleviate this problem, in this work we propose an approach that can help software engineers to more easily and safely select the proper design pattern, for a given design problem. The approach relies on decision trees, which are constructed using domain knowledge, while options are conveyed to software engineers through an Eclipse Theia plugin. To assess the usefulness and the perceived benefits of the approach, as well as the usability of the tool support, we have conducted an industrial validation study, using various data collection methods, such as questionnaires, focus groups, and task analysis. The results of the study suggest that the proposed approach is promising, since it increases the probability of the proper pattern being selected, and various useful future work suggestions have been obtained by the practitioners.

1. Introduction

Software patterns correspond to established solutions to common software development problems. In the literature, various types of patterns have been introduced: e.g., Analysis Patterns (Fowler, 1996), Architectural Patterns (Buschmann et al., 1996), and Design Patterns (Gamma et al., 1995). Among these pattern catalogues the most popular is the one introduced by Gamma, Helms, Johnson, and Vlissides (known as Gang of Four—GoF) to propose design solutions to object-oriented design problems. The GoF design patterns have been heavily studied in the academic literature, from various points of views (e.g., effect on quality, applicability, automated detection, etc. (Mayvan et al., 2017)(Ampatzoglou et al., 2013), but are also considered as a "must-have" knowledge in the software industry and are part of many software engineering curricula world-wide. Despite their wide-adoption, using GoF patterns form a skill that is not a trivial one, and the proper application and instantiation of a pattern cannot be taken for granted, especially by novice software engineers. The main problems that are faced in the adoption of patterns, are summarized below:

[p1] ***pattern selection***. Selecting the most fitting pattern for every occasion is not always straightforward. Within the pattern catalogues, some patterns can be considered as alternatives, and the discriminating line between them in some cases is very thin. For instance, consider the `Strategy` and the `Template Method` patterns. Both patterns make use of polymorphism to capture and design the different behavior (e.g., `gameLoop`) of different types of objects (e.g., `Chess` and `Backgammon`), belonging to the same general category (e.g., `BoardGames`). For the general case, the aforementioned problem can be efficiently solved with the `Strategy` pattern; however, in the special case that the different behavior shares the same skeleton / ordering of steps (e.g., `initializeBoard`, `checkIfGameIsOver`, `changeTurn`, `selectMove`), but each step is differently implemented; then, the most fitting pattern is `Template`, since `Strategy` would have led to duplicated code.

[p2] ***unnecessary use of patterns.*** In some cases, it is reported that novice developers are so eager to use a pattern, that end-up to use patterns in cases when the requirement to be implemented does not match the pattern goal. At the extreme case, there are reported cases when a developer might be keen to introduce a pattern without analyzing whether a pattern is needed in the first place. For instance, suppose the case that a number of objects needs to be created. The objects belong to some categories and subcategories, but the creation of the object is trivial and the code that instantiates the objects is not expected to be changed in the future. The use of `Abstract Factory` or `Method Factory` patterns would be feasible, but probably it will add needless complexity to the design (Martin, 2003).

[p3] ***improper instantiation.*** In some cases, the coding of a GoF design pattern solution is not so obvious and specific details need to be considered. For instance, consider the `Singleton` pattern, which requires the existence of a `self-instance` reference to the unique object of the class, a `private constructor`, and a static `getInstance` method that will first check if the single instance has been created: if not, it will create a new one, if it exists it will return the pre-created self-instance. Such

details and deviations from the standards of object-oriented programming, might not be completely clear to novice software engineers, leading to coding errors, deviations from the expected pattern instantiation, and inability to exploit the mechanism of the pattern. For example, in the case of the `Singleton` pattern, the use of a static method is important to provide a global point of access to the unique instance.

Given the above, in this work, we aim to alleviate the aforementioned problems by proposing an approach and an accompanying tool that can guide software engineers in GoF pattern selection and instantiation. In particular, based on expert knowledge we have developed several decision trees that can guide the developer along Q&A walkthroughs in selecting the most fitting pattern (related to ***p1***), or advising not to use a pattern (related to ***p2***). To guide software engineers towards using the established instantiation of the pattern, the last step of the approach performs code generation (related to ***p3***) so as to avoid pattern grime (Feitosa et al., 2017). The tool has been developed as part of the SmartCLIDE project[1], is released as an Eclipse Theia[2] extension that is stored and distributed through Eclipse Research Labs `git` repositories[3].

To evaluate the proposed approach and the accompanying Eclipse Theia plugin, we have conducted an industrial validation with an SME that is independent of the SmartCLIDE project. The approach and the tool have been evaluated with respect to their usefulness, perceived benefits, and usability. To achieve data triangulation and avoid bias, various data collection methods (such as focus groups, task analysis, questionnaires, etc.) have been used. The rest of the paper is organized as follows: in Section 2 we set the scene for this work, by presenting related studies. In Section 3, we present the proposed approach and the Eclipse Theia extension. The industrial validation study protocol is presented in Section 4, whose findings are presented and discussed in Section 5. To conclude the paper, in Section 6 we present threats to validity and final remarks in Section 7.

[1] https://www.smartclide.eu/

[2] Eclipse Theia is the cloud version of the Eclipse IDE: https://theia-ide.org/

[3] https://github.com/eclipse-researchlabs/smartclide-design-pattern-selection-theia

2. Literature Review

2.1 Design Patterns

In the late 1970s an architect named, Cristopher Alexander, attempted to find and document proven quality designs in the construction industry. In order to find them, he studied different constructions that served the same purpose and categorized them by the elements that they had in common and named them design patterns. In 1987, Kent Beck and Ward Cunningham first talked about design patterns in software engineering and by the mid 90s the concept became established and widespread in object-oriented programming(Arvanitou, 2011). According to Arvanitou, 23 patterns are listed and they are used for solving common software design problems.

Design Patterns have been categorized into three categories, and each one of them defines different problems. They are categorized into creational, structural and behavioral patterns.

- **Creational Patterns**: They are referred to standard ways of dynamically creating new objects at the execution time of a program. Their main goal is to make the code that is being used by some objects independent of the classes that define these objects, according to the open-closed design principle for proper object-oriented design. Namely, creational Patterns are the Abstract Factory, Factory Method, Builder, Prototype and the Singleton.
- **Structural Patterns**: They are related to standard ways of dynamically creating objects that reuse existing class hierarchies and this category includes Adapter, Bridge, Composite, Decorator, Facade, Flyweight and Proxy pattern.
- **Behavioral Patterns**: This category divisions the responsibilities into different classes and defines the communication between their objects at the execution time of a program. Behavioral Patterns are the Chain of Responsibility, the Command, the Mediator, the Memento, the Observer, the State, the Strategy, the Template Method and the Visitor(Arvanitou, 2011).

2.1.1 Factory Method

Factory Method belongs to the Creational Design Pattern category and provides an interface for creating families of related objects without specifying their class. The pattern's methods are responsible for creating a new object of the suitable type and for returning a reference of this object. In addition, the declared return type of the methods is the general interface of the pattern, in order for the external program to receive the requested type of object and it also needs to know each generated data type that implements the interface. Given that the program is closed to possible extensions and only the implementation of Factory Method pattern is opened to them, as only its own code needs to be modified in case e.g. adding a new class produced(Arvanitou, 2011)

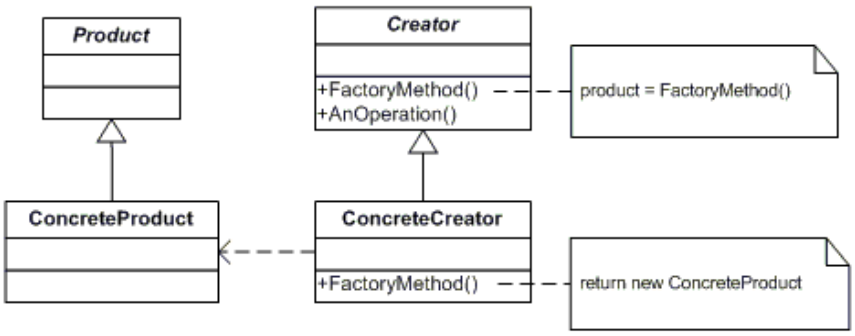


Diagram 1: Class Diagram of Factory Method Design Pattern

2.1.2 Builder

Builder is a Creational Design Pattern that allows the construction of a complex object, by using the same code for different representations of it. The procedure of construction involves putting the construction "materials" step by step. Additionally, using the Builder pattern is a way for construction details to be hidden from the client, since the `Director` is responsible for the implementation of the different representations. Director is the class that contains a variety of methods that each of them calls the right `ConcreteBuilder` (depending on the requirement of the client). `ConcreteBuilders` implement the Builder interface, in order for elimination of repeated code. `ConcreteBuilders` complete the construction of the products and, finally, the products are ready to be fetched by the `Director`.

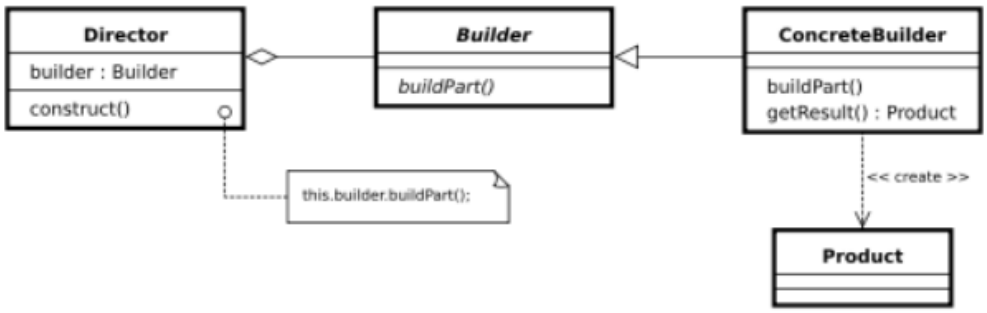


Diagram 2: Class Diagram of Builder Design Pattern

2.1.3 Bridge

Bridge is a Structural Design Pattern that is used for organizing classes into two separate hierarchies, which can be developed independently of each other. As GoF book mentions, the two hierarchies are `Abstraction` and `Implementation` (or else `Implementor`), not in a programming language, but in a conceptual one. `Abstraction` is a high-level class that controls some entity. It is not supposed to do any business logic on its own. When an operation is needed, `Abstraction` delegates the work to the `Implementation`. The `Implementation` is a lower-level interface, which is responsible for a variety of primitive operations.

`Abstraction` and `Implementation` may have Concrete Classes (`RefinedAbstractions` and `ConcreteImplementors`) in order for different entities to combine and use different functionalities.

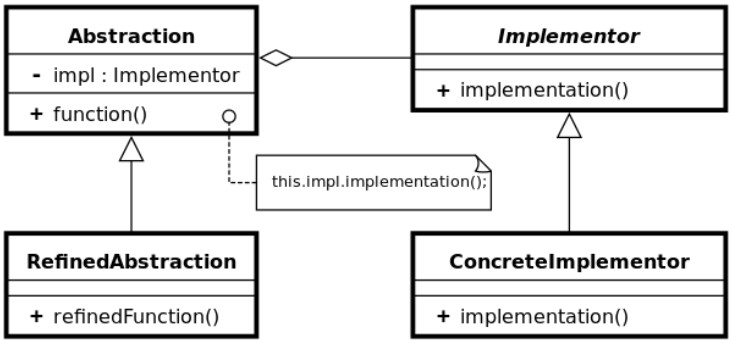


Diagram 3: Class Diagram of Bridge Design Pattern

2.1.4 Composite

Composite is a Behavioral Design Pattern that allows the objects to be composed in tree structures aiming to represent the hierarchy. The advantage of the Composite pattern is that the client does not need to know whether the objects in the tree are composite or simple. The client can treat them all the same via a common interface: `Component`. The `Component` has two subclasses: the `Leaf` and the `Composite`. `Composite` delegates all the work to its child components (`Leaves` or other `Composites`), until the child becomes a `Leaf`. Finally, `Leaf` executes all the work that needs to be done.

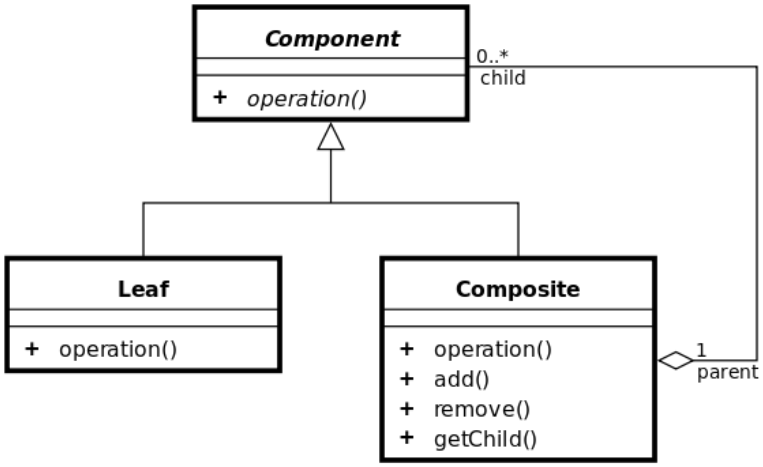


Diagram 4: Class Diagram of Composite Design Pattern

2.1.5 Command

Command is a Behavioral Design Pattern that is used when someone needs to encounter requests, in a certain way, where a class is created for each request. Command pattern can implement a queue of requests, where requests are served according to the time they arrive.

`Sender` (or else `Caller`) is the class that initiates the requests. Nevertheless, it does not create the command (request) object. The command object is pre-created in the constructor of the `Client`. Command is a simple interface that usually obtains only a method for execution (`execute`()). `ConcreteCommands` are responsible for the business logic that is required, whether to implement it or pass it to some other class (with implementation of business logic). Each `ConcreteCommand` has a field (or more), which is used as a parameter to the class method. Finally, the `Receiver` class can be almost any class, which receives the outcome of the executed command.

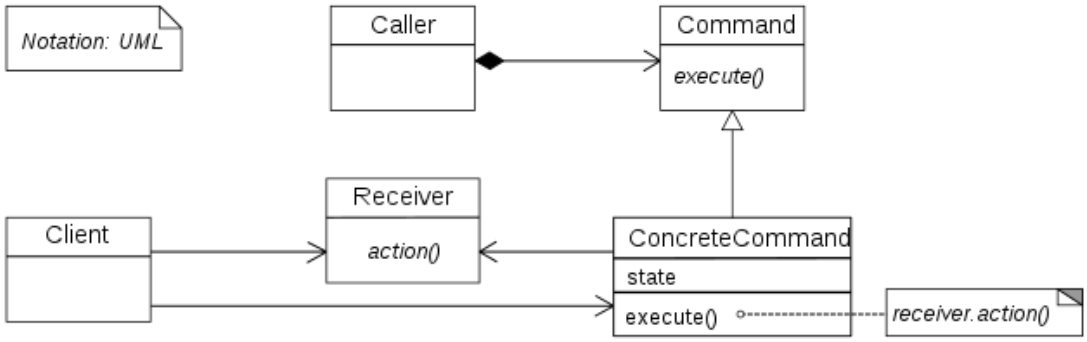


Diagram 5: Class Diagram of Command Design Pattern

2.1.6 Memento

Memento is a Behavioral Design Pattern that stores the object's internal state and allows the object to restore his state and return to the last one. Except in the case that the object wants to restore its state, this pattern is used when an interface does not want to expose implementation details and violate the object's encapsulation, when it wants to get the state of an object.
The Memento defines a class called `Memento` that is responsible for saving the state of another object class called A and for restricting the access to class A by other classes. The class A is known as `Originator` and is capable of creating an object of the class `Memento`, that is being used to restore the previous internal state. Therefore, the `Memento` pattern has another class that is capable of capturing the originator's state and keeping track of the `Originator`'s history by storing them in a list. The `Caretaker` class fetches the first `Memento` object from the list, when the `Originator` class wants to take the previous state and then passes it to the `Originator`'s method(Refactoring Guru, 2020).

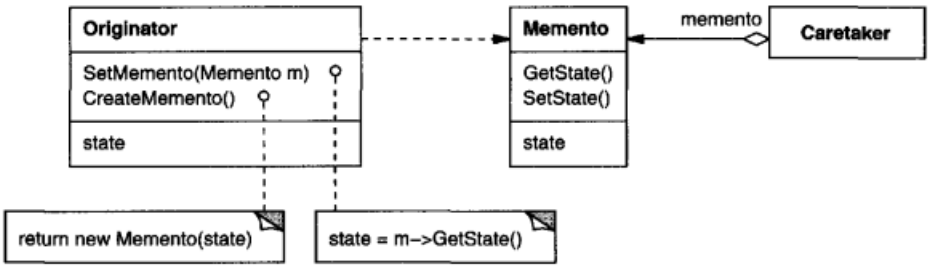


Diagram 6: Class Diagram of Memento Design Pattern

2.1.7 Strategy

Strategy is a Behavioral Design Pattern that encapsulates the state of an object so that it can alter its behavior when the internal state of the object changes. When there are different ways of solving a problem (e.g different algorithms), it is more prefered for each one of them not to be implemented in the client classes that use it(e.g as a private method), intending to lessen the complexity in the client classes and to make the to the various algorithms reusable and to be accessible by multiple programs.
The Strategy defines separate classes for the different algorithms that implement a common interface A and the clients store a reference to the interface A in a field. This field is given a value by the parameter of some method of the client (possibly its constructor) , so that the type of the algorithm that is used by all those classes to perform the requested task can be easily configured, with a simple change of this argument when calling the method.

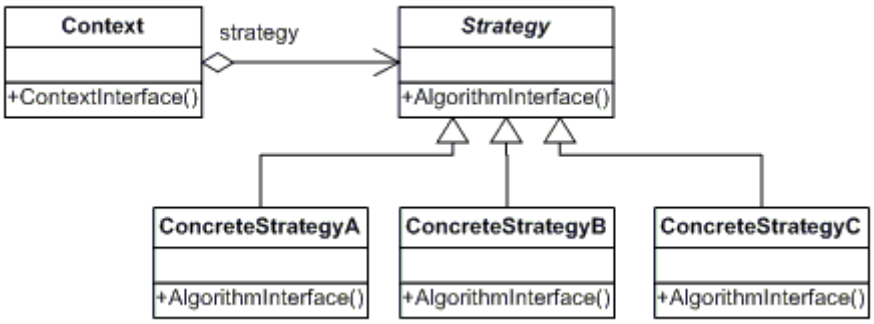


Diagram 7: Class Diagram of Strategy Design Pattern

2.1.8 Observer

Observer is a Behavioral Design Pattern and it is mainly used when there are some objects called `Observers` that are interested in receiving notifications about any changes that may occur at the state of another object called A. There are two approaches to implement these updates: either the object A calls on the predetermined methods of the `Observers` when an update of the state happens or the `Observers` invoke a method of object A, in order to receive the notifications. In the first case, only the object A is aware of the changes in his state and these changes are going to trigger the updates. One way to implement this approach is to store the `Observers`, who are interested, in a linked list and call their appropriate method at the right times. However, this approach causes the closure problem , where some other type of `Observer` who does not implement the methods that the other `Observers` implement can not be stored in the list.
The Observer pattern solves this problem by providing an Observable interface, which contains a register method and an unregister method, and an Observer interface with a synchronized method for transferring the updated data. The class of each object A must implement the Observable interface and the class of each `Observer` must implement the Observer interface. An Observable object maintains a list of Observers that have called the register method of that object. In addition, list of `Observers` may change during the execution of the program.

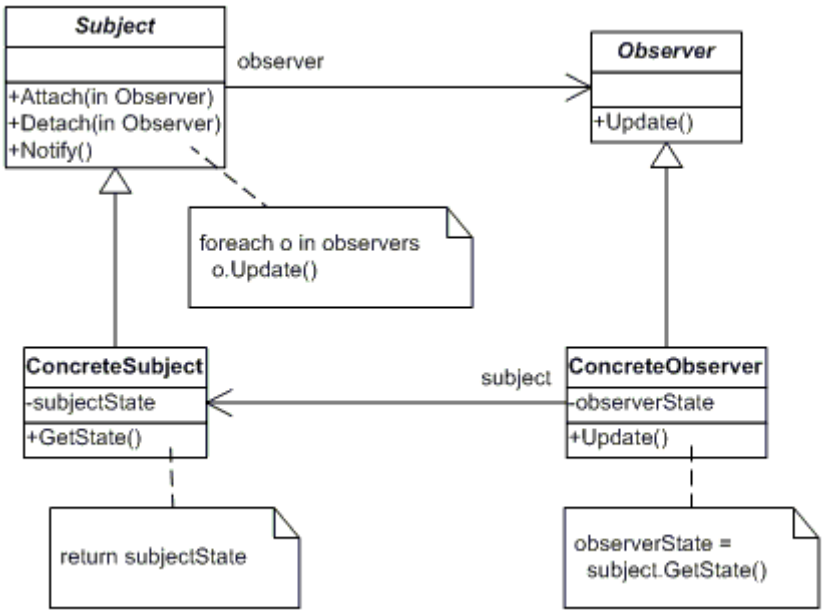


Diagram 8: Class Diagram of Observer Design Pattern

2.2 Pattern Selection Approaches

In this section we present existing studies that aim at aiding in the adoption of GoF design patterns in practice, focusing particularly in the pattern selection process. According to Ampatzoglou et al. (2013), "***GoF design pattern application"*** is the research sub-topic that involves research endeavors that present methods for identifying systems that need pattern application or methods and tools that automate or assist the application of patterns. The research landscape in this direction can be organized into 5 main categories: (a) design pattern abstraction; (b) re-engineering to patterns; (c) generative design patterns; (d) automated code transformation; and (e) pattern-based architecture. Each one of the aforementioned lines of research are described in detail below.

According to Ampatzoglou et al. [7], "***GoF design pattern application"*** is the research sub-topic that involves research endeavors that present methods for identifying systems that need pattern application or methods and tools that automate or assist the application of patterns. The research landscape in this direction can be organized into 5 main categories: (a) design pattern abstraction; (b) re-engineering to patterns; (c) generative design patterns; (d) automated code transformation; and (e) pattern-based architecture. Each one of the aforementioned lines of research are described in detail below.

The ***Design Patterns Abstraction*** research topic includes studies (e.g., [21][80][160]) suggesting that design patterns are the key to provide abstraction in software and for adapting software components into existing systems. Bishop (2008) presents how the use of the more abstract features of a programming language can decrease the gap between design patterns and their implementation. More specifically, Bishop (2008) used as examples three design patterns (i.e., Bridge, Prototype and Iterator). Design patterns presents some of their own abstraction challenges: (a) the traceability of a design pattern is hard to maintain when programming

languages offer poor support for the underlying patterns, (b) design patterns are used and reused in the design of a software system, but with little or no language support, developers must implement the patterns again and again in a physical programming language, (c) some design patterns have several methods with trivial behavior, and without good programming tools, it can be more complicate to write all this code and maintain it, and (d) using multiple patterns can lead to a large cluster of mutually dependent classes, which lead to maintainability problems when implemented in a traditional object-oriented programming language.

Keepence and Mannion (1999) develop a method that uses design patterns to model variability. The method starts by analyzing existing user requirements from systems within the family and identifying discriminants, which is any feature (requirement) that differentiates one system from another. There are three types for the identification of discriminants: (a) single discriminant, which is a set of mutually exclusive features, only one of which can be used in a system, (b) multiple discriminant which is set of optional features that are not mutually exclusive; at least one must be used, and (c) option discriminant which is single optional features that might or might not be used. The authors tested their method on ESOC's spacecraft MPSs. They built a family user-requirement specification by editing and merging the requirement specifications from three separate MPSs: ISO (a spacecraft that observes stars), ERS-2 (a remote-sensing spacecraft that monitors the earth's environment, and Cluster (a multi-spacecraft mission to monitor the earth's magnetosphere). The family user-requirement specification had 350 requirements (each MPS requirement specification had about 150 user requirements). Based on the analysis of the MPS family, they produced 20 class diagrams, 15 object-interaction diagrams, and 100 classes. This model lets developers identify and select desired features and build new family systems.

Yau and Dong (2000) present an approach to apply design patterns to component integration. This approach uses a formal design pattern representation and a design pattern instantiation technique of automatic generation of component wrapper from design pattern. Design patterns are organized in a design pattern repository, where patterns are represented precisely using their design pattern representation. The design pattern representation should be expressive without jeopardizing the abstract feature of the design pattern solution. Components and their descriptions can be retrieved from a component repository. The component description includes component interfaces expressed in IDL and semantics of services provided by components. After the selection of the design pattern, the pattern has to be instantiated to a concrete solution. Design pattern instantiation is to generate part of the software design, based on the generic solution in design pattern and application-specific pattern instantiation information. Finally, while applying design patterns, the designers should ensure the consistency between the original design patterns and the instantiated design patterns. The approach is assessed using an illustration example. The example is to develop a chatting room, which is used for several people in one group to talk simultaneously.

The ***Re-engineering Anti-Patterns*** research topic includes studies that propose methods for detecting software anti-patterns that necessitate re-engineering through design pattern application. Briand et al. (2006) present a structured methodology for semi-automating the detection of areas within a UML design of a software system that are good candidates for the use of design patterns. This is achieved by the definition of detection rules formalized using the OCL and using a decision tree model. More specifically, each tree corresponds to a design

pattern (e.g., Decorator) or a group of design patterns when those patterns have strongly related structures and intent (e.g., Factory Method and Abstract Factory). Decision nodes in a tree denote a question in the decision process towards the identification of places in the design where design patterns could be used. When a series of questions have been answered, the tree leads to a decision where a design pattern is suggested. This corresponds to a path in the tree, from the root node to a leaf node. Additionally, some of the decisions are semi-automatic and involve user queries. Moreover, the authors illustrate their methodology using the Factory Method and the Abstract Factory Design Pattern. The aforementioned methodology has been implemented in a tool namely DPATool (Design Pattern Analysis Tool). The DPATool consists of three sub-systems: (a) the DPA Eclipse plugin, (b) the DPA Processing Engine, and (c) the DPA Model. The DPATool is a plugin to the Eclipse platform that interacts with two other Eclipse plugins, namely the Eclipse UML2 and Eclipse EMF plugins. The tool could be used by two different types of users. First, expert designers, who can define their own decision trees, for instance according to their observations of how designers in their organizations develop system. Second, every designer can be invoked whenever necessary during UML-based development support by Eclipse. To assess the feasibility of their methodology, they performed a case study of a test driver for an ATM. The ATM test driver has 15 classes with 114 operations and 45 attributes. The UML 2.0 models of the ATM test driver were reverse-engineered from the source code into the Eclipse platform. After processing the UML 2.0 model of the ATM test driver, the DPATool suggests the usage of a Factory Method pattern. Also, DPATool suggested the use of the Visitor and the Adapter design patterns.

Meyer (2006) provides an approach, which supports the detection of anti-pattern implementations in source-code. More specifically, the approach consists of three main steps: (a) anti-pattern recognition, (b) transformation, and (c) transformation verification. For the first step, the approach is based on an extended Abstract Syntax Graph (ASG) representation of a system's source-code. Anti-patterns are specified by graph grammar rules, which define as an ASG node structure which has to exist in the ASG representation and adds an annotation node to indicate the anti-pattern. The approach parses the source-code into the ASG representation and the anti-pattern rules are applied to the ASG by an inference algorithm. For the transformation step, the transformation rules are specified as graph grammar rules based on Story Diagrams. The software engineer manually examines the candidates identified by the first step and decides which transformations are to be applied to which candidate, if any. Then, the transformations are executed automatically in the transformed source-code. As the final step, the transformation rules must verify that the rules do not create forbidden or preserved anti-patterns.

Generative Design Patterns correspond to techniques that aim to automatically generate design pattern instances. MacDonald et al. (2002) present an approach to generative design patterns, trying to solve three problems: (a) there are no adequate mechanism to understand the variations in the source-code that spans the family of solutions and adapt the code for a particular application, (b) it is difficult to construct and edit generative design patterns, and (c) the lack of a tool independent standard. Their approach is independent of programming language and support tools. To validate the approach, they have implemented two tools, CO2P2S (Correct Object-Oriented Pattern-based Programming System) and MetaCO2P2S to support the process. The process consists of three steps. First, the software engineer selects an appropriate

generative design pattern from a set of supported patterns. Second, he / she adapts this pattern for their application by providing parameter values. Finally, the adapted generative pattern is used to create object-oriented framework code for the chosen pattern structure. In a follow-up study, the same group (MacDonald et al., 2009) presents a design-pattern-based programming system based on generative design patterns that can support the deferral of design decisions where possible, and automate changes where necessary. Moreover, a generative design pattern is a parameterized pattern form that is capable of generating code for different versions of the underlying design pattern. Also, the author categorized the design decisions into two categories: (a) interface-neutral decisions—affect only the implementation of the structure of the pattern behind a stable interface, and (b) interface-affecting decisions—affect both the structure of the pattern and the framework interface to the application code. CO2P3S (Correct Object-Oriented Pattern-based Parallel Programming System, pronounced "cops") generates Java frameworks for several common parallel structures, both shared-memory code using threads and distributed-memory code. The author demonstrated the capability of the system in the context of a parallel application written with the CO2P3S pattern-based parallel programming system.

The ***transformation to pattern*** research topic includes studies that propose methodologies for automatically constructing transformations that can be used to apply GoF design patterns. O' Cinneide and Nixon (2001) present a methodology and tool support, namely DPT (Design Pattern Tool), for the development of design pattern transformations. The methodology deals with the issues of reuse of existing transformations, preservation of program behaviour and the application of the transformations to existing program code. First, a design pattern is chosen that will serve as a target for the design pattern transformation under development. Then, the transformation is decomposed into a sequence of mini-patterns (i.e., a design motif that occurs frequently across the design pattern catalogues). For each mini-pattern, a corresponding mini-transformation (i.e., an algorithm that applies the corresponding mini-pattern to the given program entities) is developed. Then, each mini-transformation should be demonstrated as a behaviour-preserving. The algorithm that describes the mini-transformation is expressed as a composition of refactorings. The final design pattern transformation can be defined as a composition of mini-transformations. The authors used the Factory Method transformation as an illustrative example. Moreover, the authors present a prototype software tool DPT that has been designed and implemented that can apply these pattern transformations to a Java program. Finally, they used an example of the application, the Factory Method transformation to a generic program. The authors applied the methodology to a set of patterns from the GoF catalog, and prototyped the transformations. For each pattern, first the method finds a suitable precursor, assessing if a workable transformation can be built, and determining the mini-transformations that are likely to be used. Then, the authors assessed the results based on the three categories (excellent, partial, and impractical). The results suggest that half of the patterns have excellent transformation and 26% of the cases as partial.

Hsueh et al. (2010) provide an approach for design pattern application and support the design enhancement by model transformation. For the selection of the pattern for the model transformation, the authors divided the pattern into six parts: pattern description, functional requirement intent, non-functional requirement intent, functional requirement structure, non-functional requirement structure, and transformation specification. For the automating pattern application, Hsueh et al. (2010) document the refinement processes of patterns in regular rules

and describe them in formal transformation language. Then, after specifying the transformation specification, they implement the mapping rules in ATLAS Transformation Language (which is a hybrid of declarative and imperative transformation language based on OMG OCL). For the evaluation of their approach, the authors performed a case study on a real-world embedded system PVE (Parallel Video Encoder). They define the Command Pipeline pattern to revise a sequential processing design to a parallel processing design in a generative TBB code.

Finally, Tonella and Antoniol (2001) propose an approach for documenting design decisions in real-time, and enables ***pattern-based architecture*** through the inference of object-oriented (OO) design patterns from the source-code or the design. As a first step, the authors have used concept analysis to identify groups of classes sharing common relations. Next, the selected concepts contain maximal collections of classes having the same relations among them. The aforementioned concepts seem to be good candidates to represent design patterns inferred from the source-code or from the design. The number of instances of a pattern represents an indicator of the frequency of reuse of the identified class organization, while the number of involved relations represents the complexity of the pattern. To evaluate their approach, Tonella and Antoniol (2001) performed a case study on C++ applications. They first examined the methods that were owned by the involved classes. The results of their study suggest that the structural relations among classes led to the extraction of a set of structural design patterns, which could be improved with non-structural information about class members and method invocations.

2.3 Benefits of Using Patterns

Software Design Patterns organize and structure the code in a way that the final software gains quality. This quality can be described and categorized in two groups: internal and external. The internal quality is composed of extendibility, understandability and reusability (Ampatzoglou, 2012).

According to Ampatzoglou et al (2015), maintainability is a crucial quality concern of the research community that investigates the impact of GoF design patterns on software quality attributes. Studies that put maintainability as a priority for research count to 40% and show that GoF design patterns provide a framework for maintainability.

Additionally, Ampatzoglou mentions that, according to research, 18 out of 23 design patterns positively affect the *maintainability* of the software (Ampatzoglou, 2012). Particularly, the survey took place with highly experienced software developers on GoF design patterns. They were given eight quality features and all the GoF patterns and they were asked to evaluate the latter by the former. Maintainability was mainly connected with expandability (Ampatzoglou et al., 2015).

More specifically, a study of Prechelt et al., that took place in 2001, included tasks given to professional software engineers about five of the GoF patterns: Abstract Factory, Observer, Decorator, Composite, and Visitor. Tasks were supposed to be completed with and without using the GoF patterns. The study concluded that using a pattern than a simpler solution is usually preferable. Vokac et al., in 2004, tried a similar study, but this time in a real programming environment. The results suggest that the call for applying a GoF pattern must be made by the designer's judge (Ampatzoglou et al., 2015).

Other studies that mentions Ampatzoglou et al. (2015) includes a study that refers to the understandability and the modifiability of Visitor design pattern instances. Student participants

were asked to do some comprehension and modification tasks on open-source projects with canonical and non-canonical representations of the Visitor pattern. The outcome of the study suggests that the effort, which the modification tasks on the canonical representation needed, was less than the one for the tasks on the non-canonical representation. The outcome gets enhanced when the participants have knowledge of UML notations. Another study on complexity, coupling, cohesion and size metrics suggests that the implementation of GoF design patterns improves cohesion, coupling, and complexity of the systems. However, there are some disadvantages: the increased number of code lines and number of classes. Moreover, a survey that investigates the effects of GoF patterns on maintainability resulted in enhanced maintainability when an architectural pattern has been used compared to alternatives (Ampatzoglou et al., 2015).

Maintainability is connected with the Open Close Principle. A study has taken place in order to conclude whether the implementation of State pattern would result in the enhancement of the code's conformance to the Open Closed Principle. Finally, the study showed that only in a percentage of 20%, the conformance to the Open Closed Principle is possible without the implementation of the pattern. As an outcome, if we consider the Open Close Principle to be the main way of maintaining the Object-Oriented way and protecting the addition of subclasses (instead of modification of the existing ones), then there is only a 20% chance for a system to achieve all the above without an instance of State pattern (Ampatzoglou et al., 2015).

A quality characteristic that has an ambiguous effect is *reusability*, because some patterns are easier to be reused. As for *understandability*, it is considered to be the most ambiguous quality characteristic, since a few patterns like Visitor, Composite, Decorator, Proxy, Observer and Abstract Factory, appear to be easy-understood by some researchers and hard-understood by others (Ampatzoglou, 2012).

It is expected patterns, such as Facade, Flyweight, Mediator and Memento to amplify software reusability. Nevertheless, the reinforcement depends on the context of the implementation. For example, Flyweight is considered to be difficult in understanding and generalizing. Software mechanics avoid implementing it into their code, because, in order for this to succeed, the problem (that needs to be solved) must be known. Similar criticism has occurred for Mediator. However, when Mediator is used for connecting subsystems, it raises the possibility of class reusability, since subsystems are independent and easily detached and adaptable (Ampatzoglou, 2012).

The benefits of using patterns are not always 100% present, because patterns have both pros and cons. The only way to make a calculation of the outcoming quality is to consider and assess the context and the requirements of the problem. Usually, when a quality characteristic appears reinforced, then another seems to fade. So, if a problem requires flexibility, a pattern that offers flexibility, but increases complexity, should be accepted and used. For example, Abstract Factory is known for contributing to the maintainability of the system, as new types of products can be added without any changes being necessary to the code. Nevertheless, using Abstract Factory might reduce the understandability of the code. In such cases, the designer of the code needs to prioritize the quality characteristics and choose the design pattern that serves the priorities in the best way (Ampatzoglou, 2012).

In this paragraph, there will be presented the structural features of some design patterns that were used in this research and their impact on internal quality characteristics:

- *Builder*: uses polymorphism in its methods in order for creation of product families. So, Builder enhances polymorphism.
- *Bridge*: As mentioned in paragraph 2.1.3, Bridge separates the control class (Abstraction) from the lower-level Implementation class. This is succeeded by polymorphism and class inheritance, by avoiding nested control commands. This mechanism, on one hand, offers low complexity, loose coupling, high cohesion and high-level hierarchies. On the other hand, it raises the declarative overhead, because many extra classes and class methods are produced.
- *Command*: Every type of command, as mentioned in paragraph 2.1.5, creates a Concrete Command class. As a result, the size of the system becomes enlarged.
- *Strategy*: uses polymorphic methods in order to eliminate control commands. As an outcome, methods become less complex and coupling more loose. However, the size of the system increases, because of the great number of classes. Additionally, the various behavior of the objects comes from different classes, resulting in high cohesion of the system.
- *Observer*: uses a polymorphic behavior in order to update a hierarchy of classes in a united way. This way, the system separates the observer classes from the classes that are observed.

Research on external quality from using design patterns is less extended and highly recommended for future projects (Ampatzoglou, 2012).

3. Proposed Solution

3.1 Pattern Selection Decision Trees

The proposed approach for assisting software engineers in selecting GoF patterns is based on binary decision trees, i.e., sequences of questions that involve binary answers, and gradually exclude irrelevant patterns, or pin-point to the most fitting ones. The methodology to construct the binary decision trees involved various iterations among pattern experts from both academia and industry:

- study the definitions and examples of GoF patterns from various sources, e.g., books (Gamma et al., 1995) (Shalloway & Trott, 2004) and online sources[4,5]
- compile sets of patterns that are alternatives, and a primary reason that leads to the selection of a pattern
- review the aforementioned outcomes, by pattern experts from academia and industry partners
- group the reasons to use a pattern, with most common reasons being closer to the root of the decision tree
- transform the reasons to a Q&A format
- review the obtained decision trees by pattern experts in four rounds of feedback and update of the decision tree. In each round after the first, and additional expert was added. The review rounds were terminated when the additional expert had no supplementary feedback.

3.1.1 Creational Patterns Selection Decision Tree

Below, we demonstrate how the aforementioned process has been applied for the case of Creational Design Patterns. We note that for simplicity only the outcomes of reviewed steps are being demonstrated, since intermediate outcomes would only cause disruption to the reader.

Reasons to Apply:

`Abstract Factory`: Create New Object, Create Different Types of Products, Families of Products Exist

`Factory Method`: Create New Object, Create Different Types of Products

`Builder`: Create New Object, Create Different Types of Products, Product can be Produced in Steps

`Singleton`: Reuse a Unique Object of a Specific Class for the whole project instead of Creating New

`Prototype`: Create Copies of a Specific Class Objects instead of Creating New

Grouping of Reasons (in Coloring Scheme):

`Abstract Factory`: Create New Object, Create Different Types of Products, Families of Products Exist

`Factory Method`: Create New Object, Create Different Types of Products

[4] https://refactoring.guru/design-patterns

[5] https://sourcemaking.com/design_patterns

`Builder`: Create New Object, Create of Different Types of Products, Product can be Produced in Steps
`Singleton`: Reuse a Unique Object of a Specific Class for the whole project instead of Creating New
`Prototype`: Create Copies of a Specific Class Objects instead of Creating New

The aforementioned grouping leads to a 4-level decision tree. The 1st level, differentiates between the creation of a new object and the reuse / clone of an existing objects (instead of creating a new one)—red vs. blue fonts: "*Do you want to create a NEW object or to reuse an existing one?*". Following the red font criterion, the next criterion (green fonts) is common for all alternatives (2nd level—left part): "*Does the product has sub-categories*?". Thus, if it is not fulfilled by targeted requirement, then NO pattern shall be used. Next, we need to select for the final level specific criteria (black fonts), we opted to first ask "*Can the products be classified to a family of products?*" (3rd level—left part), and then "*Can a product be created in a series of steps?*" (4th level—left part). By following blue criterion at the 1st level, we have two distinct questions. We have selected to ask: "*Do you want the object to be cloned or unique?*" (2nd level—right part). The aforementioned rationale, is depicted in Figure 1. A similar way of working has been performed for Structural Design Patterns (see Figure 2) and Behavioral Design Patterns (see Figure 3). We note that in the decision trees of Figures 1-3, apart from the aforementioned Q&A, we also have some questions on the class names that will play the role for each pattern. This part has enabled the code generation for a specific project. The notation for reading Figures 1-3 is as follows: (a) Green rectangles represent the questions responsible for pattern selection; (b) Blue rectangles represent questions that aim at gathering information for code generation; and (c) Red ovals correspond to outputs of the process. The available responses out of each green rectangle are designated on the relative arrows leaving the node.

3.1.2 Structural Patterns Selection Decision Tree

Reasons to Apply:

`Bridge`: Use information from two different hierarchies
`Facade`: Communicates with Existing Artifact, Communicates with Subsystem
`Flyweight`: Communicates with Existing Artifact, Communicates with One Class, Is used for Reducing Memory Usage
`Adapter`: Communicates with Existing Artifact, Communicates with One Class, the Existing class can Not Change
`Proxy`: when Access Control needed
`Decorator`: Have Composite Object, Have Layers that extend the Behavior of the Object
`Composite`: Have Composite Object

Grouping of Reasons (in Coloring Scheme):

`Bridge`: Use information from two different hierarchies
`Facade`: Communicates with Existing Artifact, Communicates with Subsystem
`Flyweight`: Communicates with Existing Artifact, Communicates with One Class, Is used for Reducing Memory Usage
`Adapter`: Communicates with Existing Artifact, Communicates with One Class, the Existing class can Not Change

`Proxy:` when Access Control needed
`Decorator:` Have Composite Object, Have Layers that extend the Behavior of the Object
`Composite:` Have Composite Object

The Selection Decision Tree of Structural Patterns is layered in 6 levels of decision nodes. The 1st level, differentiates whether the client needs to use information from two different hierarchies in order for the problem to be solved or not. The bridge pattern is quite unique and difficult to be categorized with the others, so we chose to put the question for it first. If the user responds affirmatively to the question: *"Do you need to implement a function that requires information from 2 different hierarchies?"* (black fonts), then the path will lead him to the Bridge pattern. If he responds negatively, then the question from 2nd level will show up: *"Is any of your objects a composite one (i.e. composed of simple objects), which however needs to be treated uniformly along with simple objects?"* (blue fonts). Following the blue font criteria, the next question is: *"Are there different layers that extend the behavior of the Composite object?"* (black fonts, 3rd level - right part). This question leads to the Decorator pattern, if the answer is yes, or else, to the Composite one. The left part of the same level (3rd) includes the question: *"Do you want to communicate (reuse or hide the complexity) with an existing artifact (class or subsystem)?"* (red fonts). This question is common for all the non-mentioned yet patterns and leads to the 4th level of the decision nodes. At this level there are two questions: *"Communicate with one class or subsystem?"* (green fonts, left part) and *"Do you need access control for the some service class?"* (black fonts, right part). The first question is common for three patterns: Facade, Flyweight and Adapter. A positive answer leads to Facade pattern, while a negative answer leads to the 5th level of the decision nodes. The second question ends up in the Proxy pattern, except if the user chooses "No" and, in this case, there is No Pattern that can fit the user's needs. At 5th level, the showing decision question is the following: *"Do you need an interface in order to reduce memory usage?"* (black fonts). This question has been put in the tree in order to stand out the Flyweight pattern. If someone continues further, he will meet the question: "Are you unable to change the interface of the existing class?" (black fonts). At this point, all decision nodes have appeared and the user needs to choose one last time whether the question suits him or not. If it does, then the pattern that he needs is the Adapter. Otherwise, there is No such Pattern. The sequence of possible steps that just described is depicted in Figure 2.

3.1.3 Behavioral Patterns Selection Decision Tree

Reasons to Apply:

`Command:` Handle Requests, Known Recipient of the Request, Handle Requests as Objects
`Mediator:` Handle Requests, Known Recipient of the Request, Hiding the Internal Class of a Component
`Chain of Responsibility:` Handle Requests, Unknown Recipient of the Request
`Strategy:` Handle States, Varying Implementations of an Algorithm, Implement with Polymorphism
`Template Method:` Handle States, Varying Implementations of an Algorithm, Varying Implementations of a Common Algorithm

`Visitor:` Handle States, Varying Implementations of an Algorithm, Extended Implementations Based on Existed Ones

`Memento:` Handle States, Handle States of an Object (Not Algorithm), Save the State - Undo

`Observer:` Handle States, Handle States of an Object (Not Algorithm), Broadcast the State

`State:` Handle States, Handle States of an Object (Not Algorithm), Handle diverse states through inheritance

Grouping of Reasons (in Coloring Scheme):

`Command:` Handle Requests, Known Recipient of the Request, Handle Requests as Objects

`Mediator:` Handle Requests, Known Recipient of the Request, Hiding the Internal Class of a Component

`Chain of Responsibility:` Handle Requests, Unknown Recipient of the Request

`Strategy:` Handle States, Varying Implementations of an Algorithm, Implement with Polymorphism

`Template Method:` Handle States, Varying Implementations of an Algorithm, Varying Implementations of a Common Algorithm

`Visitor:` Handle States, Varying Implementations of an Algorithm, Extended Implementations Based on Existed Ones

`Memento:` Handle States, Handle States of an Object (Not Algorithm), Save the State - Undo

`Observer:` Handle States, Handle States of an Object (Not Algorithm), Broadcast the State

`State:` Handle States, Handle States of an Object (Not Algorithm), Handle diverse states through inheritance

The Selection Decision Tree of Behavioral Patterns is layered in 6 levels of decision nodes. The root question, *"Do you need an Object that will handle requests for executing an action?"* (yes: red fonts / no: purple fonts), is fundamental and common for all the patterns, because behavioral patterns handle either requests or states (of algorithms/objects), so it splits behavioral patterns into two main categories. At the 2nd level of the decision nodes, there are two questions: *"Is the recipient of the request known?"* (green fonts) and *"Do you need varying implementations of algorithms, executed under different conditions?"* (blue fonts). The positive answer of the first question of this layer leads to a 3rd level question, while a negative one leads to the Chain of Responsibility pattern. About the second question, we will reach 3rd level, whether we respond with a "Yes" or a "No". So we move to the 3rd level, which consists of three questions. The one at the left is *"Is the receiver part of a complex component, whose internal structure you want to hide?"* (black fonts), which leads to the Mediator pattern, except for the case of a "No", where a question of the 4th level will appear. The question at the middle is *"Are the varying implementations based on existing implementations, being extended in different ways?"* (black fonts), which leads to the Visitor pattern. Last and right of the 3rd level is the question: *"Do you need to manage an object with different states?"* (orange fonts), which is common for the Memento, the Observer and the State patterns and has been put there in order to inform the user that there is no other pattern than the three just mentioned for handling states

and, particularly, states of an object. Moving to the 4th level, from left to right, we meet the questions: *"Do you prefer to handle different requests as objects instead of methods?"* (black fonts), *"Are the varying implementations part of a common skeleton algorithm?"* (black fonts) and *"Do you need every state of the Object to be saved, offering the implementation of "undo"?"* (black fonts). Reasons with black font correspond to questions that have at least an edge to a non-decision node. This means for each of the three questions that one of their two routes lead to a pattern and, particularly, to the Command pattern, the Template Method pattern and the Memento pattern, respectively. The first question, otherwise, leads to "No pattern" and the others to the 5th level of question nodes. For distinguishing the Strategy pattern, a question like: *"Implement different implementations with polymorphisms?"* (black fonts - left part/5th level) was added. At the right part of the 5th level, we have placed the question: *"Do you need the change of state to be broadcasted to interested parties?"* (black fonts) for standing out the Observer pattern. If the answer to this question is negative, then the flow is transferred to the lowest decision node of the tree: *"Handle diverse states through inheritance?"* (black fonts). By following the question, we reach the State pattern. In any other case, we reach the "No Pattern" node. At this point, the presentation of the Behavioral Patterns Decision Tree has been completed and the aforementioned flow is depicted on Figure 3.

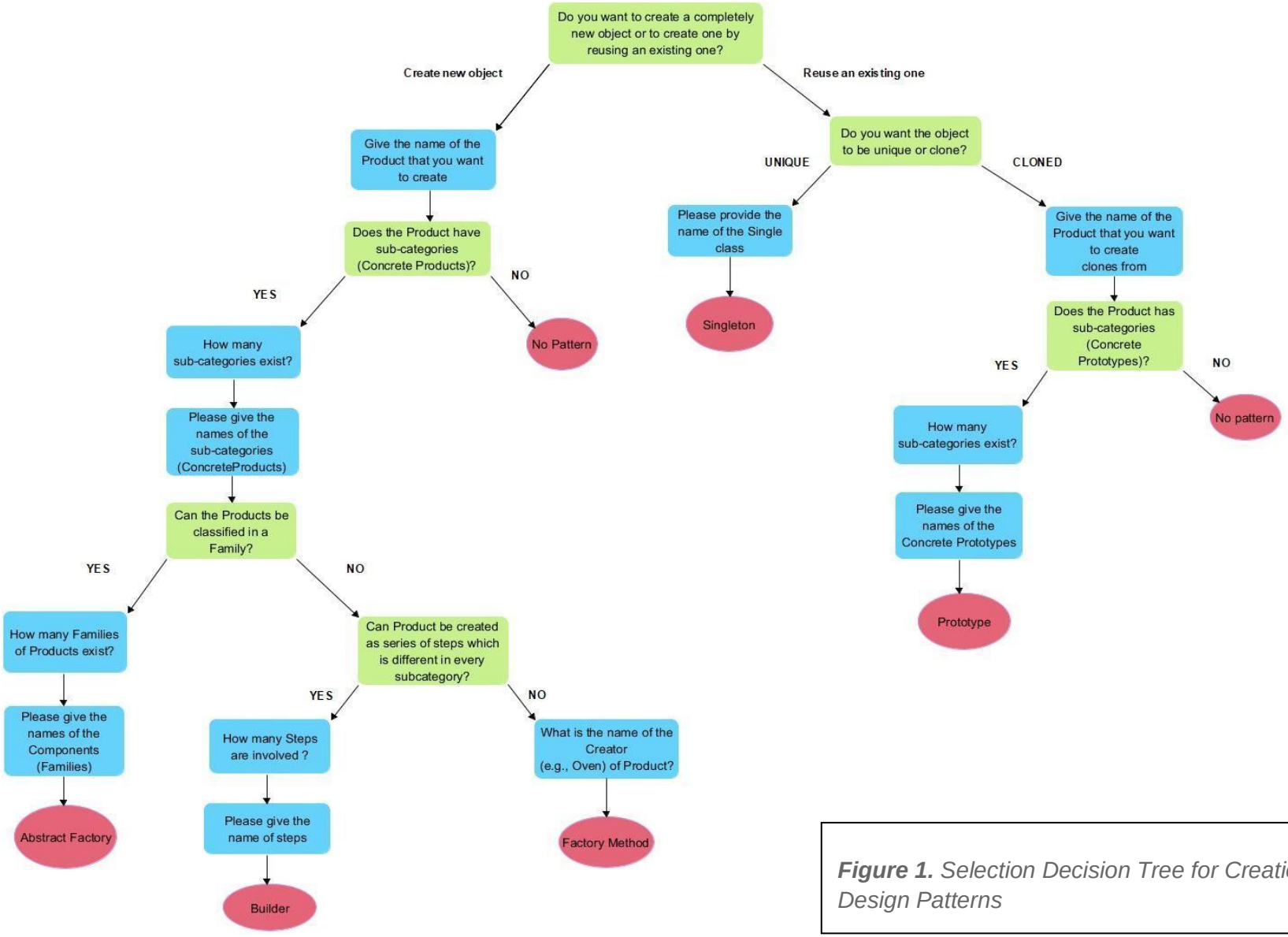


Figure 1. *Selection Decision Tree for Creational Design Patterns*

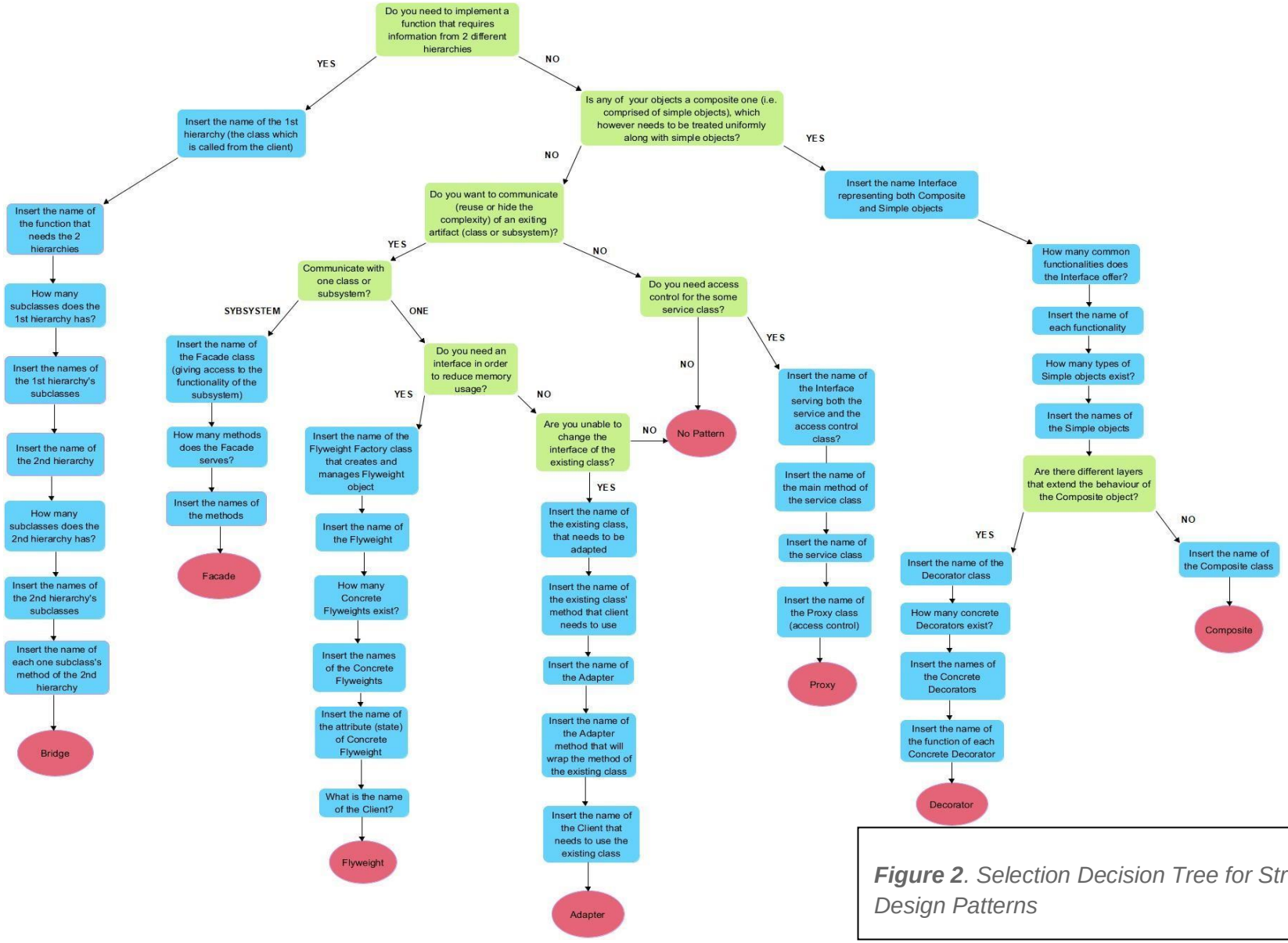


***Figure 2**. Selection Decision Tree for Structural Design Patterns*

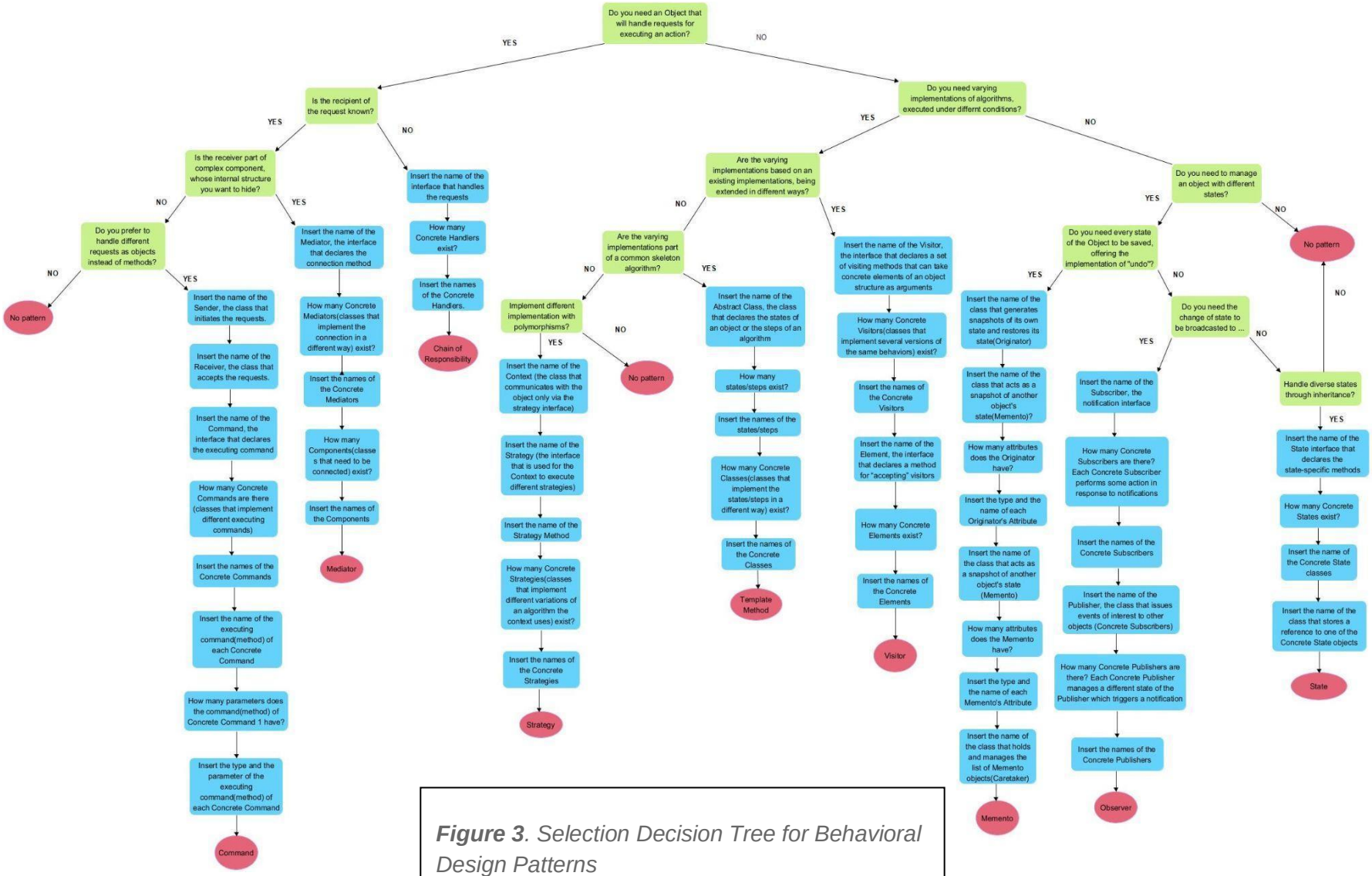


***Figure 3**. Selection Decision Tree for Behavioral Design Patterns*

3.2 Tool Support: Eclipse Theia Extension

3.2.1 Eclipse Theia

Eclipse Theia is an open, extensible and flexible framework to develop Cloud and Desktop IDE-like products with modern web technologies. Its extensibility comes from its highly modular design. There are three types of modules that extend its functionality and can be used complementary: *VS Code extension*, *Theia extension* and *Theia plugin*[8].

VS Code extension is simple at writing and can be installed at runtime. This type of extensions can be used in both Theia and VS Code. However, they have some API restrictions, because they are used for adding features to existing tools[8].

Theia extension has almost no API limitations, because it is used to build custom products. It is installed at compile time and provides full access to Theia internals via dependency injection. Theia extensions are the fundamental components of the whole Theia project, combined in a very modular way[8]. Therefore, the extension mechanism of Theia is both very powerful and flexible[9]. Someone, in order to create a Theia project, has only to choose a number of Theia extensions (core extensions) that cover his/her requirements and to combine them with his/her custom Theia extensions. If he/she compiles them and runs the result, he/she gets the almost "custom" Theia IDE. Technically, an extension is an `npm` package that exposes any number of DI modules (ContainerModule) that take part in the creation of the DI container. Extensions are consumed by providing a declaration of them as a dependency in the package.json of the application/extension[8].

Theia plugin was introduced in a later time than Theia extensions. The use of Theia plugin came from the openness of the tool, where developers needed to add features, without affecting the stability of the base tool. Also, Theia plugins are installed at runtime, without any possibilities of harming or slowing down the full product[9]. Unfortunately, Theia plugins are not compatible with VS Code.

For creating Pattern Selection Theia Extension, we used a widget. A widget is a part displaying content within the Theia workbench. Through the widget, we could place our own UI in the Theia-based application. Theia does not depend on a specific UI technology. However, it provides convenience support by providing respective base classes, React. As for the time, React is the most common choice for producing custom widgets. So, our Theia extension uses React Technology and is written in the Typescript language[10].

3.2.2 Presentation of the tool

3.2.2.1 Tool Walk-through

Upon the installation of the SmartCLIDE theme for Eclipse Theia, deployed as a Docker container[6], the user is able to open the Design Pattern Assistant from the View menu item, appearing in the left side of the screen (see Figure 4). For this demonstration we use the Apache commons-io project in a local working instance of the platform[7]. In the Figure 5, we can see

---

[6] https://hub.docker. com/repository/docker/nikosnikolaidis/theia-td-creation-patterns
[7] http://195.251.210.147:3232/#/home/project/commons-io
[8] https://theia-ide.org/docs/extensions
[9] https://eclipsesource.com/blogs/2019/10/10/eclipse-theia-extensions-vs-plugins-vs-che-theia-plugins/amp
[10] https://theia-ide.org/docs/widgets/

that the user is given two options: (a) select a pattern from the drop-down menu, if he/she feels confident on the pattern that will be used (***EXPERT-MODE***); or (b) use the WIZARD to start the Q&A process (***WIZARD-MODE***).

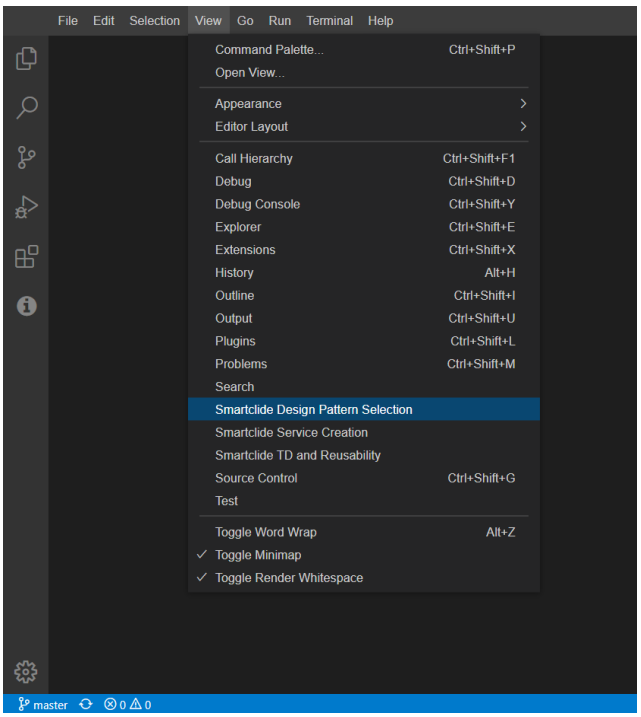


Figure 4. *Launching the Theia Extension*

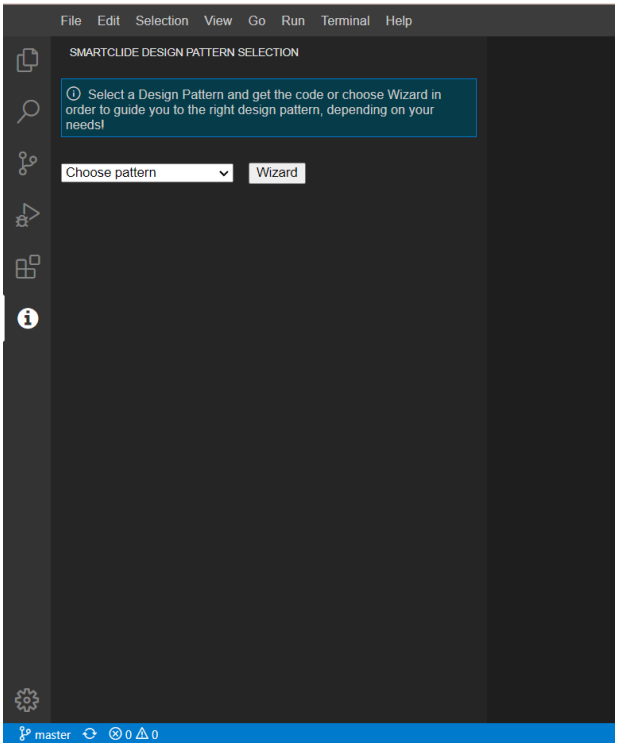


Figure 5. *Welcome Screen*

In Figure 5, we present the main layout of ***option-a***, i.e., to directly select a pattern. Having selected a GoF pattern, the software engineer is firstly reminded of the aim of the pattern, and

he/she is guided in the application of the pattern through a textual example and an accompanying class diagram (see Figure 6). In Figure 7, we present the way that the Q&A process of pattern selection appears. At the starting of the Q&A process, the user must choose the group of the pattern that he possibly needs. The advantage of ***WIZARD-MODE*** is that the paths to the design patterns can be branched. As a result, many alternatives can be created and the user has the capability to roll back to other ways. At the right part of the figure, you can see the final part, where the name of the pattern, which is suggested, appears and next to it is the button "Get Code", responsible for the Code Generation.

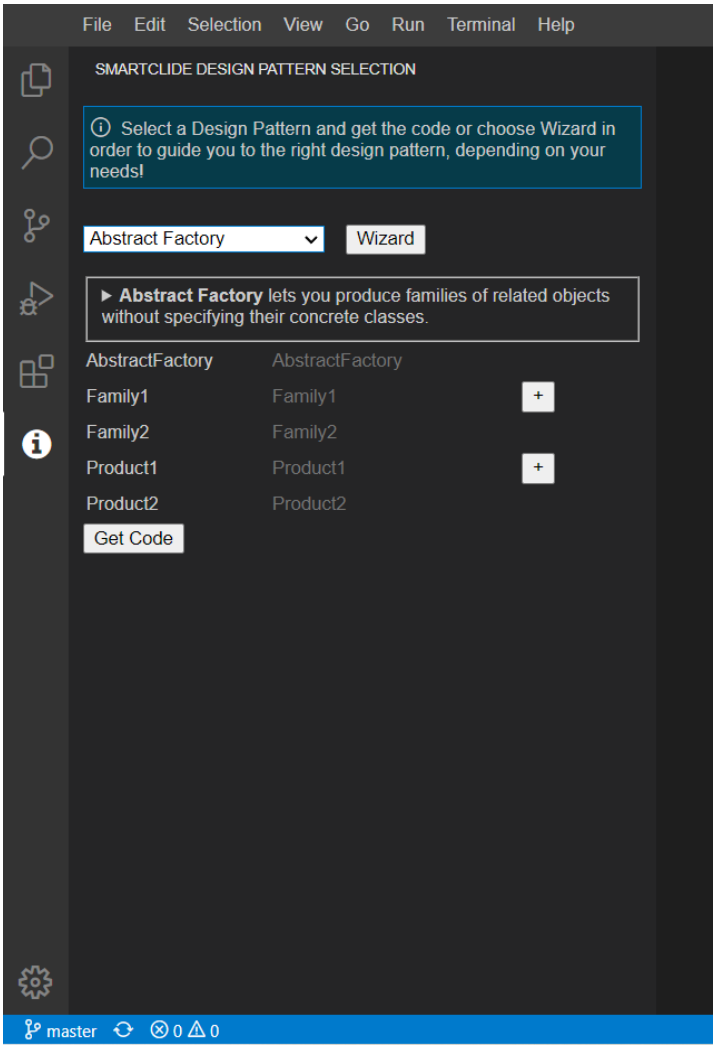


Figure 6. *Pattern Selection in EXPERT-MODE*

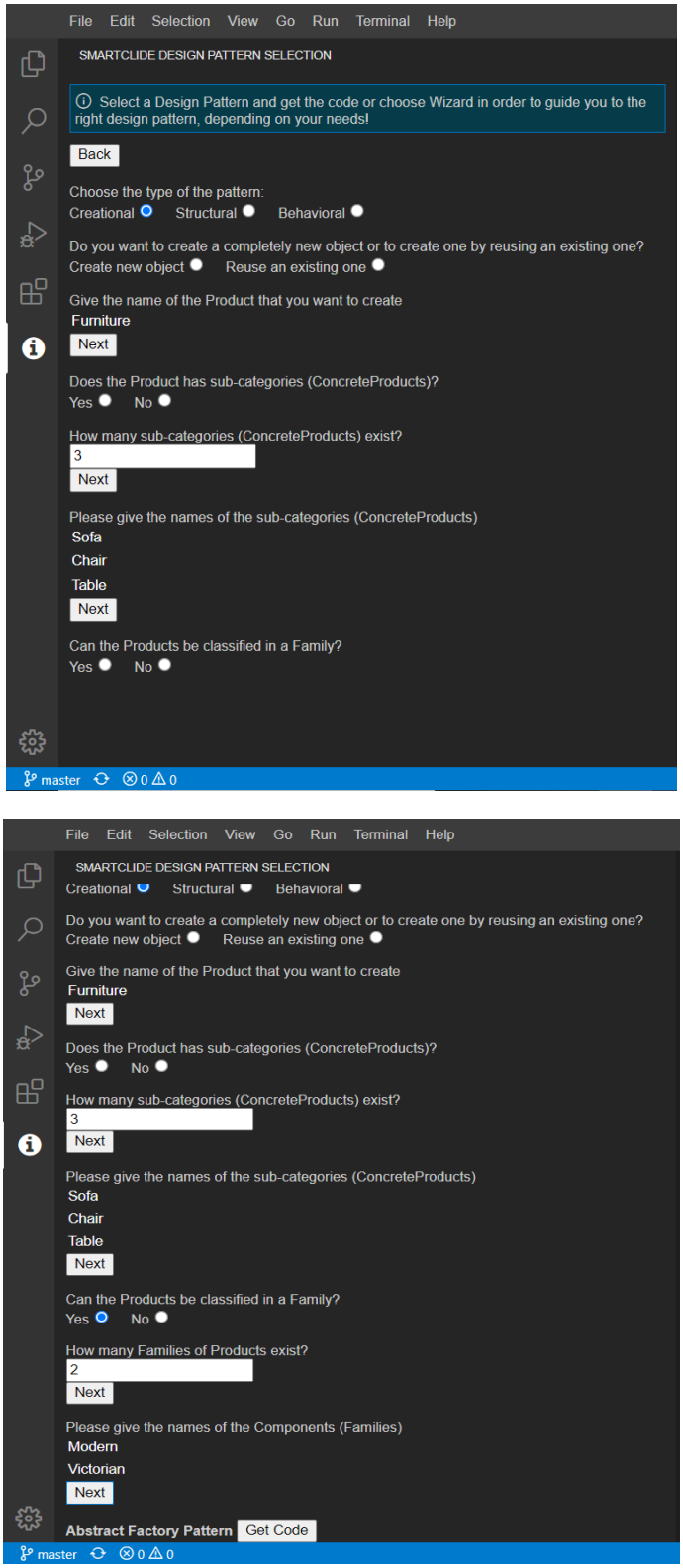


Figure 7. *Pattern Selection in WIZARD-MODE*

For both options, the roles of the pattern are mapped to either existing classes of the system, or new ones, relying on an autocomplete functionality, as presented in Figure 8.

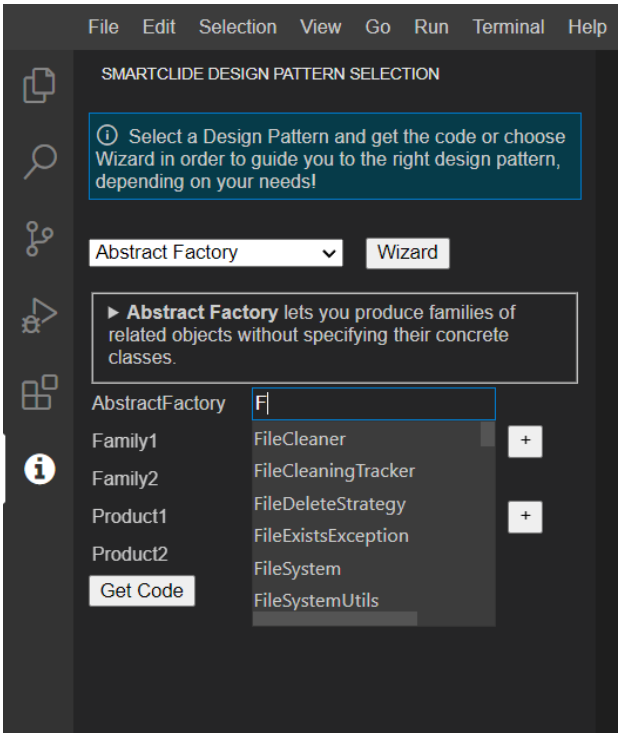


Figure 8. *Autocompleter operation*

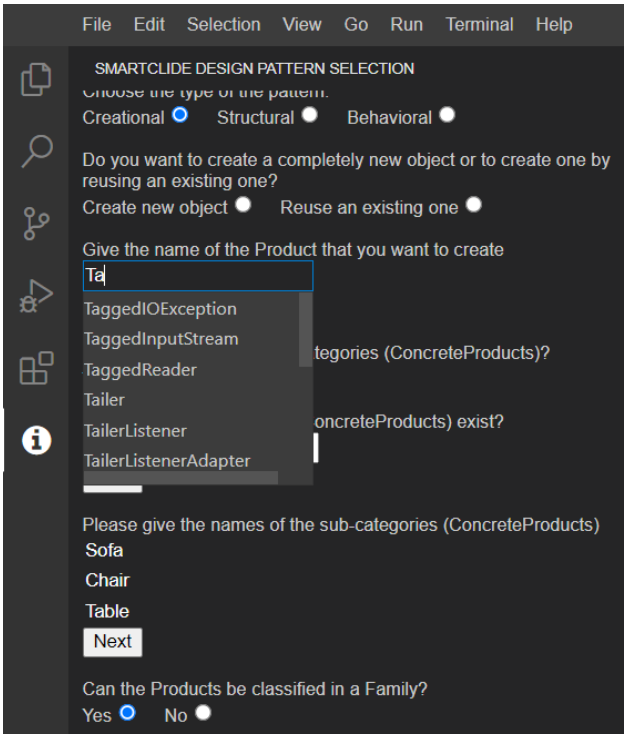


Figure 9. *Autocomplete operation*

Expect for the autocomplete functionality, at the ***EXPERT-MODE*** the software engineer can add multiple new textfield by clicking on the plus button (see Figure 9).

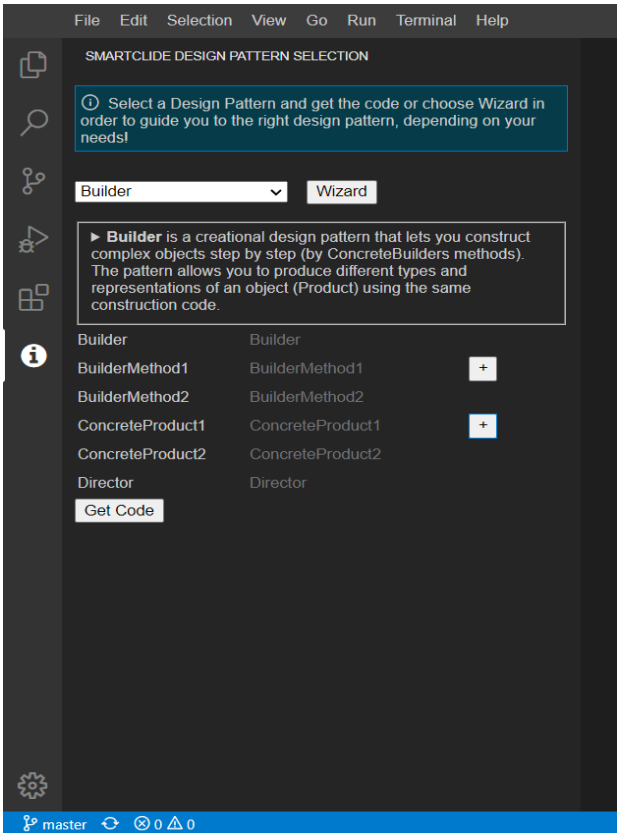
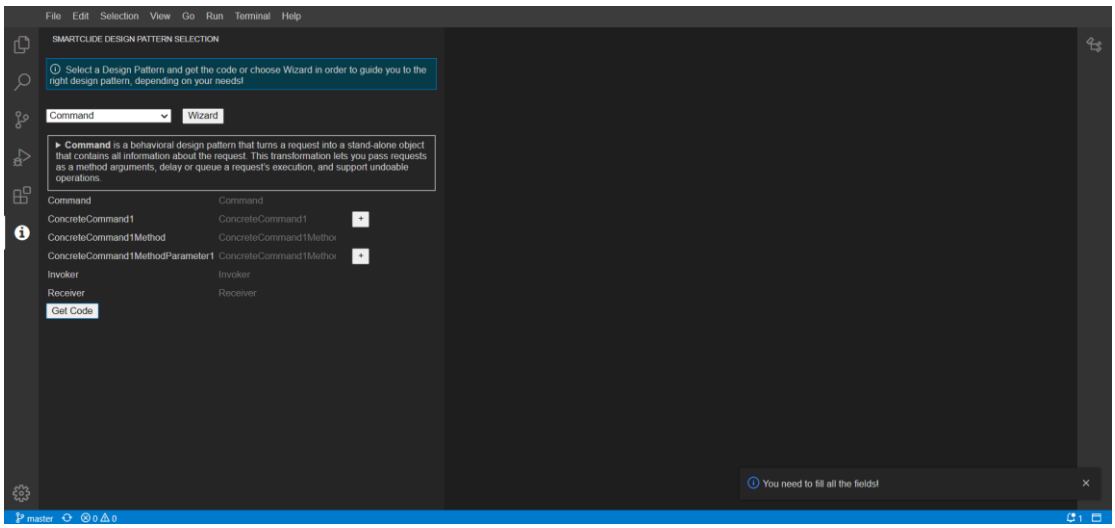


Figure 10. *Plus button operation*

In both options, there are certain controls for the input in the textfields. The input is checked, firstly, for empty values, secondly, for duplicated ones and, finally, for the writing of the entities' names. The entities refer to Class, Method, Attribute and method's Parameter. For the Class, the inserted name must follow the Pascal case, while the Method's name must follow the camel case. The names of Attributes or Parameters have to be consistent with the camel case, as well, but before the name, the type must be inserted first. So, the type and the name of the Attribute/Parameter have to be separated by a space. If any of these controls fails, then an error message appears on the right bottom of Theia, as shown in figure 10.



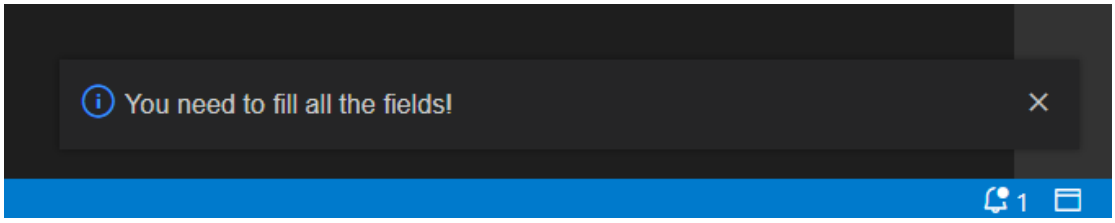
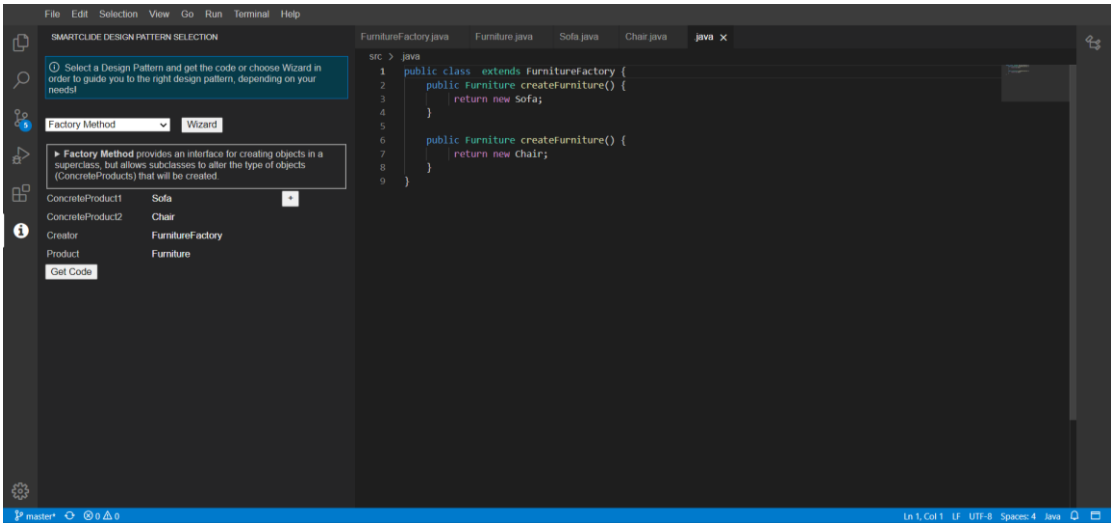


Figure 11. *Error message*

In the bottom part of Figure 11, we can see an example of generated code for the Factory Method example. For the case of using existing classes, the code of the pattern is appended in the end of the existing code, whereas for new classes the files are generated and pushed in the Git repo of the project.



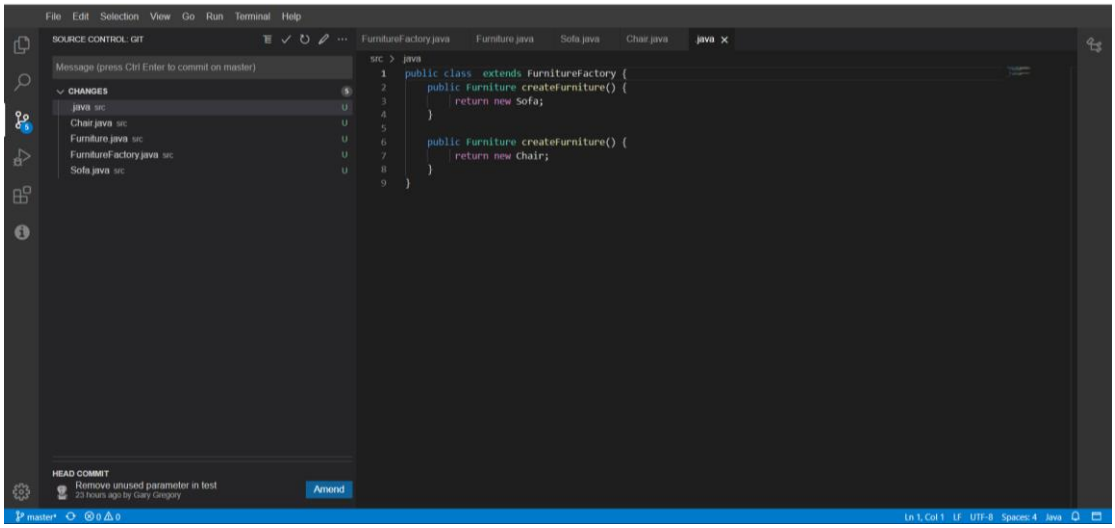
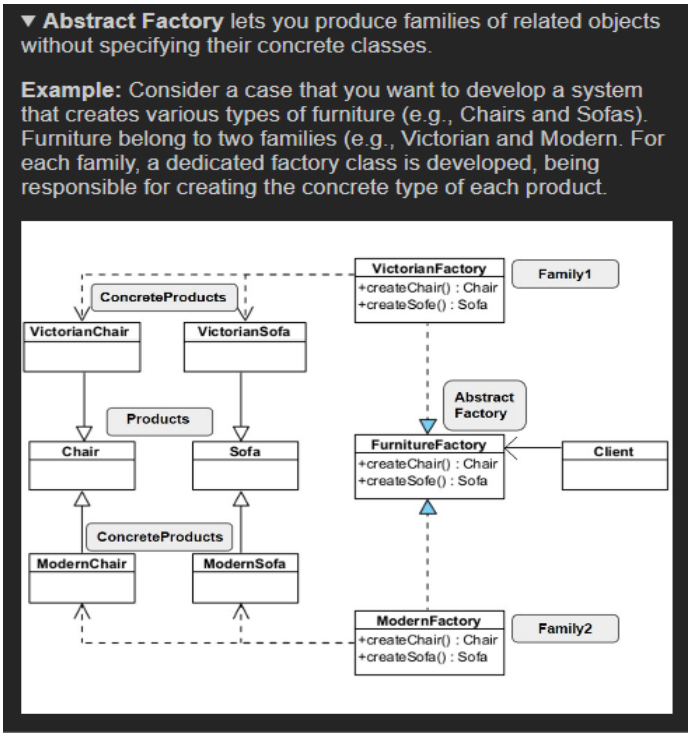


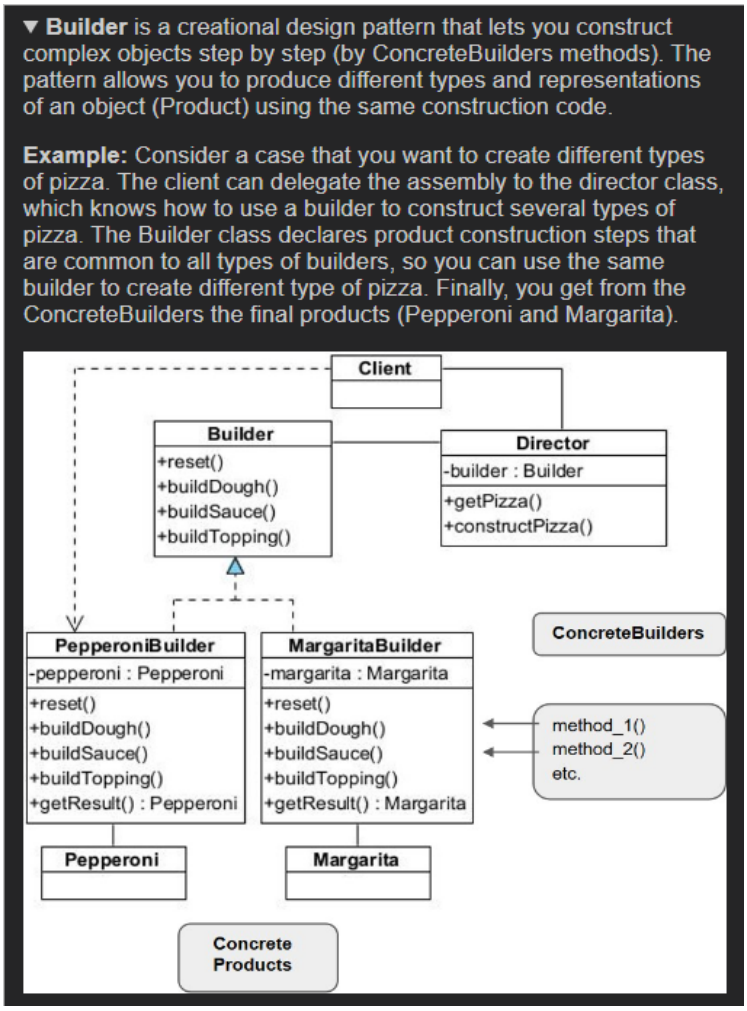
Figure 12. *Code Generation*

3.2.2.2 Pattern examples

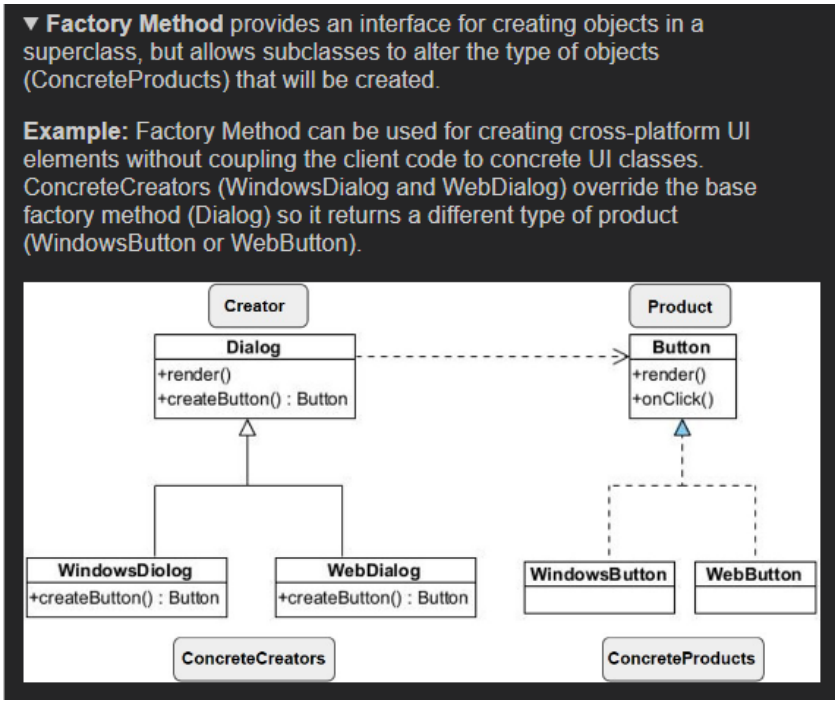
Below, we present the mini description of each pattern, that consists of the pattern's basic characteristics, an example use case and a class diagram.



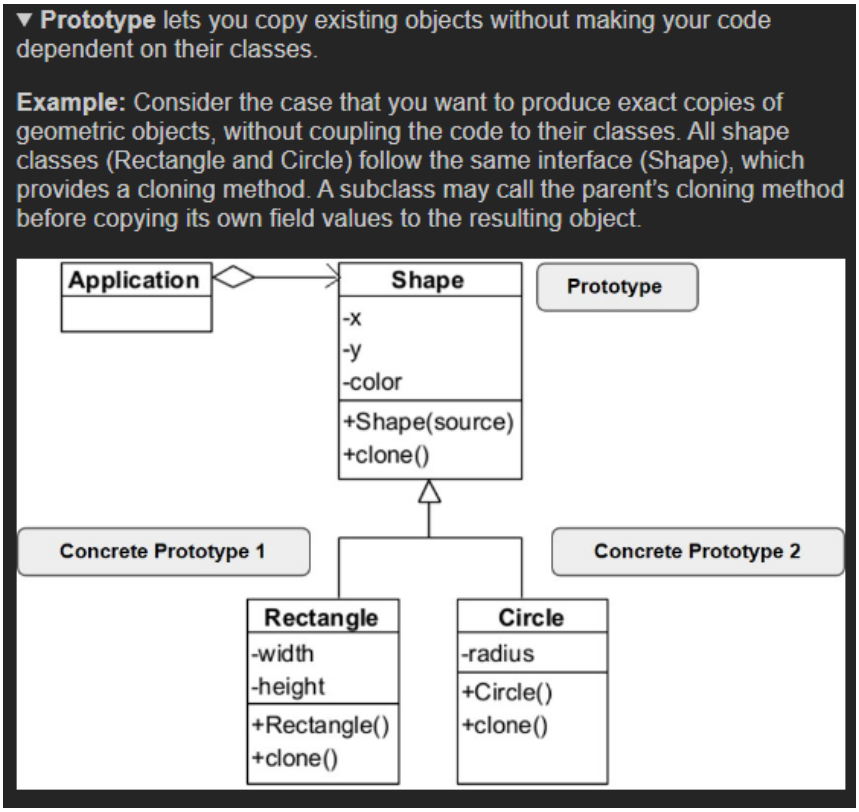
Screenshot 1. The expandable panel of Abstract Factory



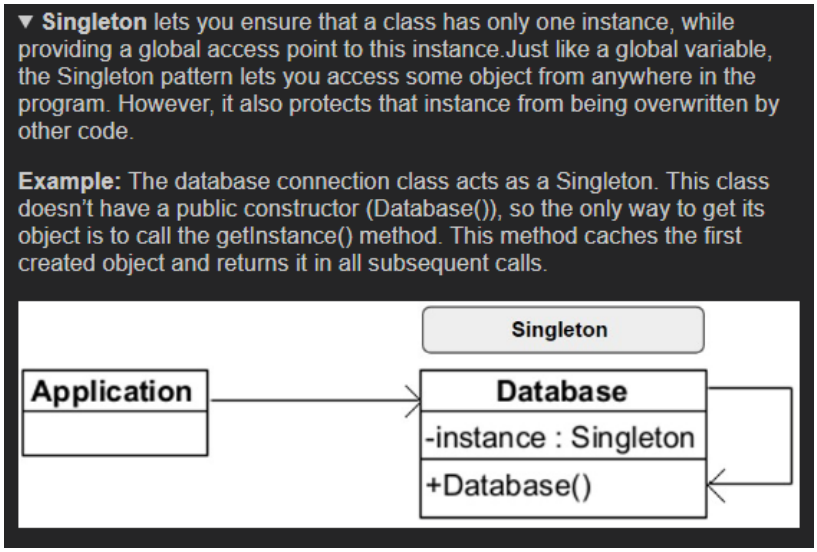
Screenshot 2. The expandable panel of Builder



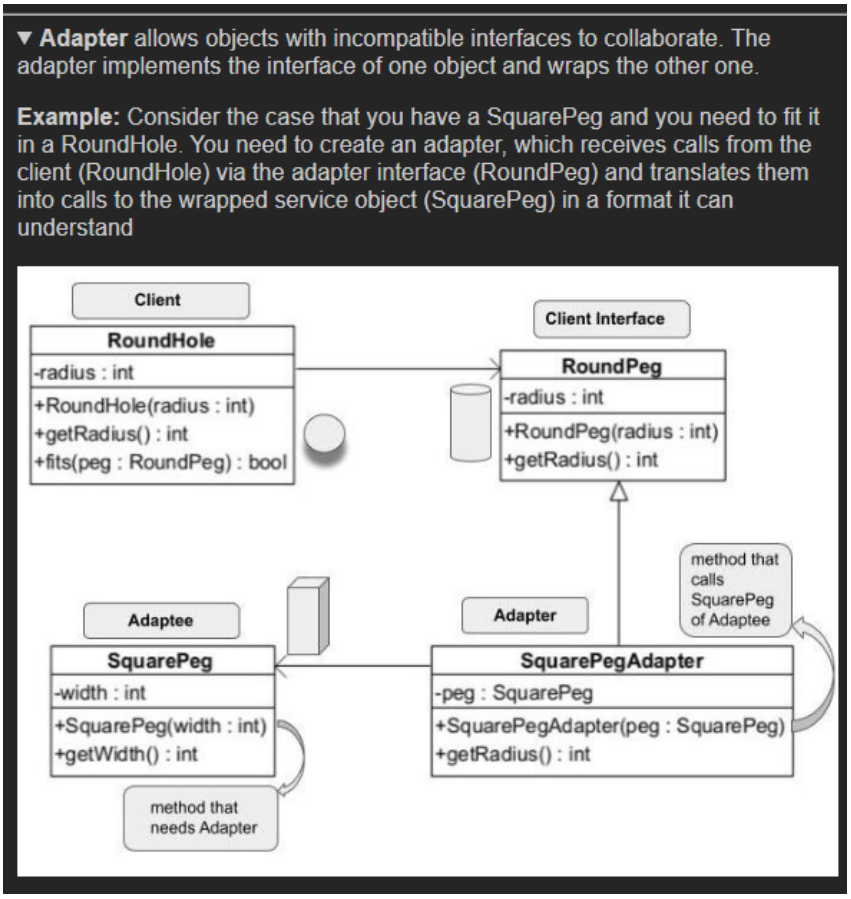
Screenshot 3. The expandable panel of Factory Method



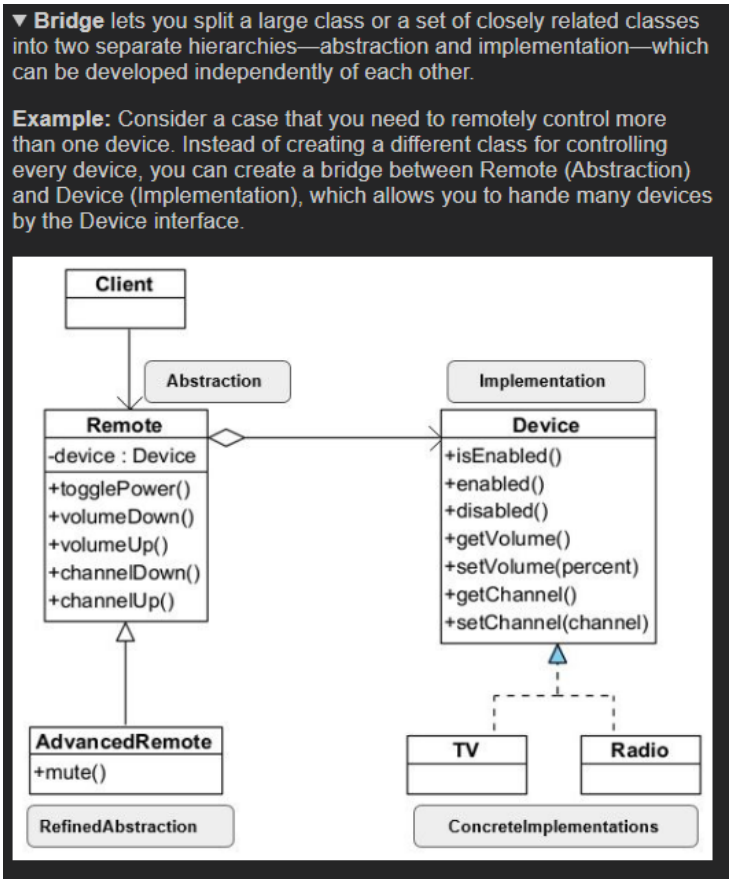
Screenshot 4. The expandable panel of Prototype



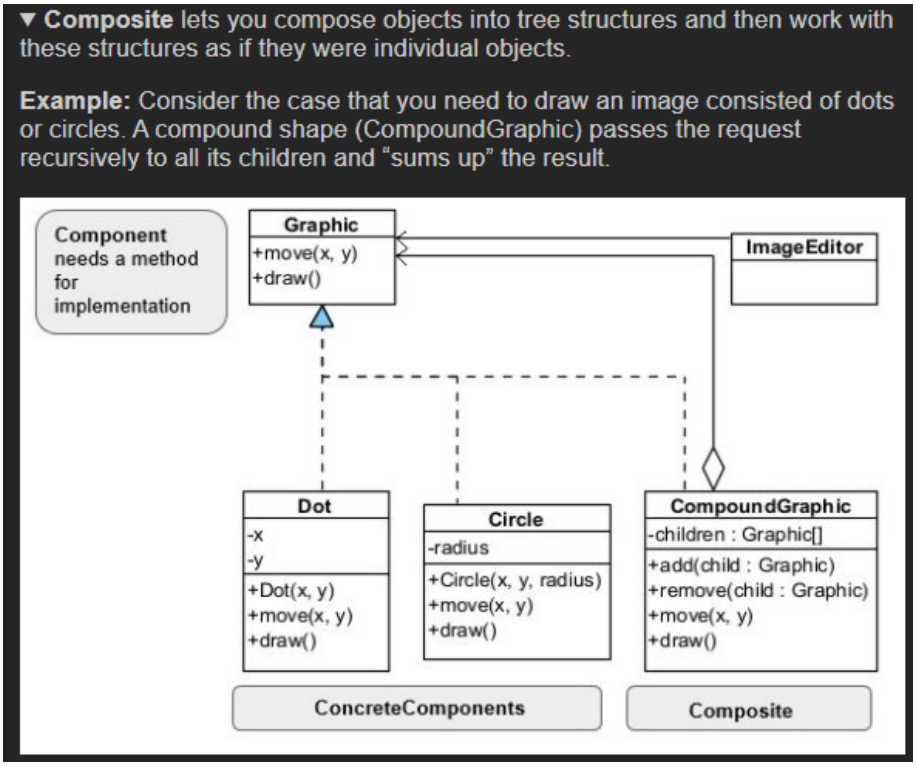
Screenshot 5. The expandable panel of Singleton



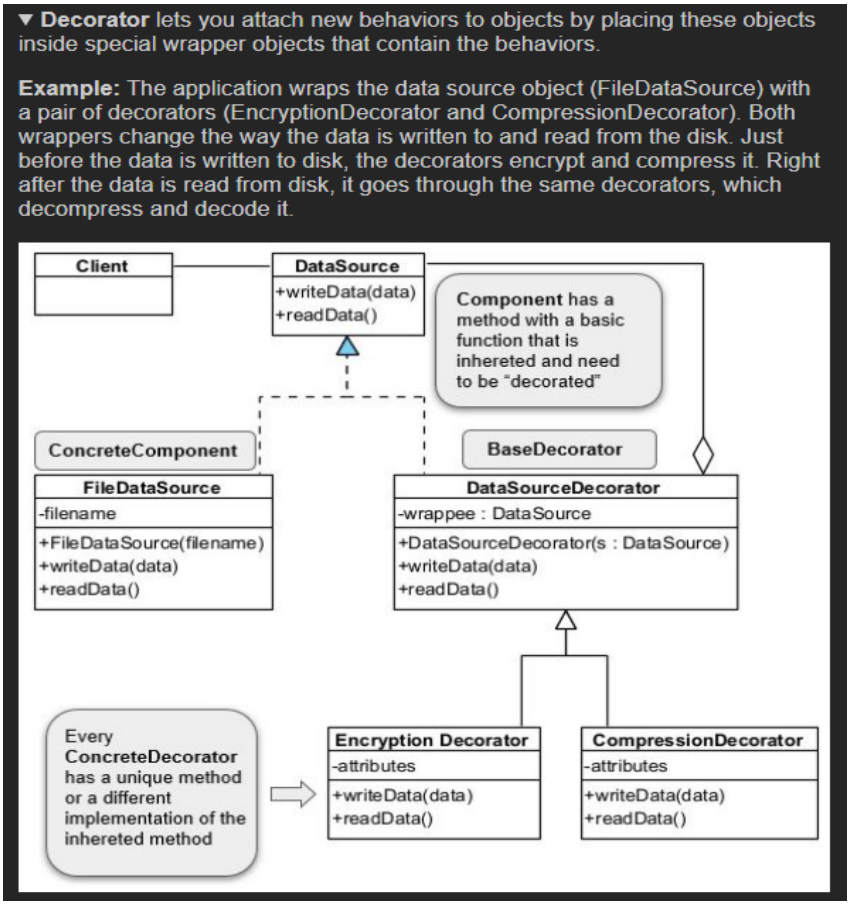
Screenshot 6. The expandable panel of Adapter



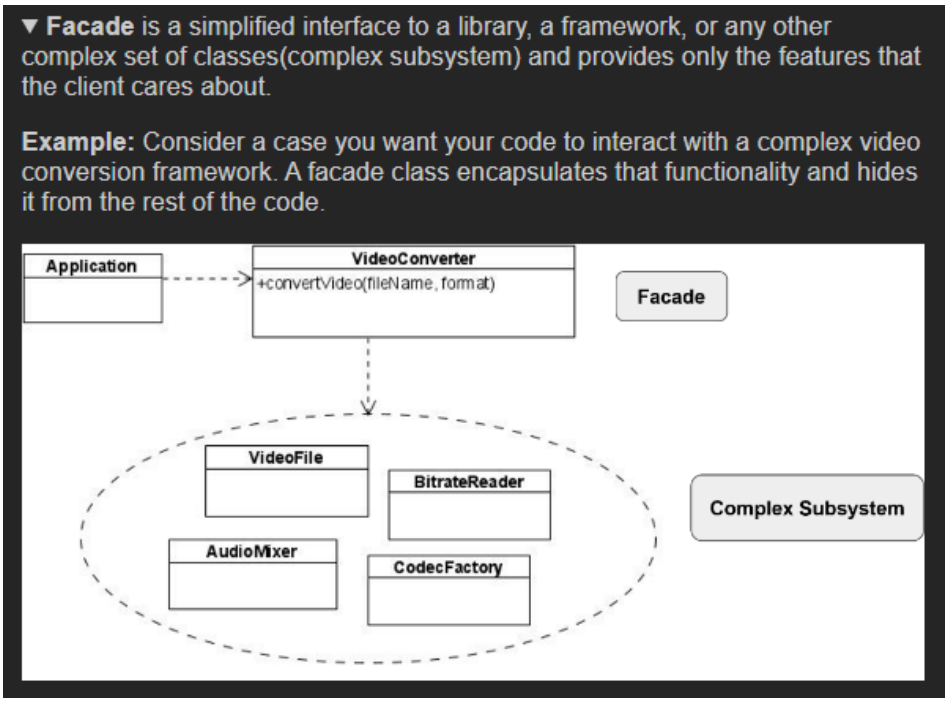
Screenshot 7. The expandable panel of Bridge



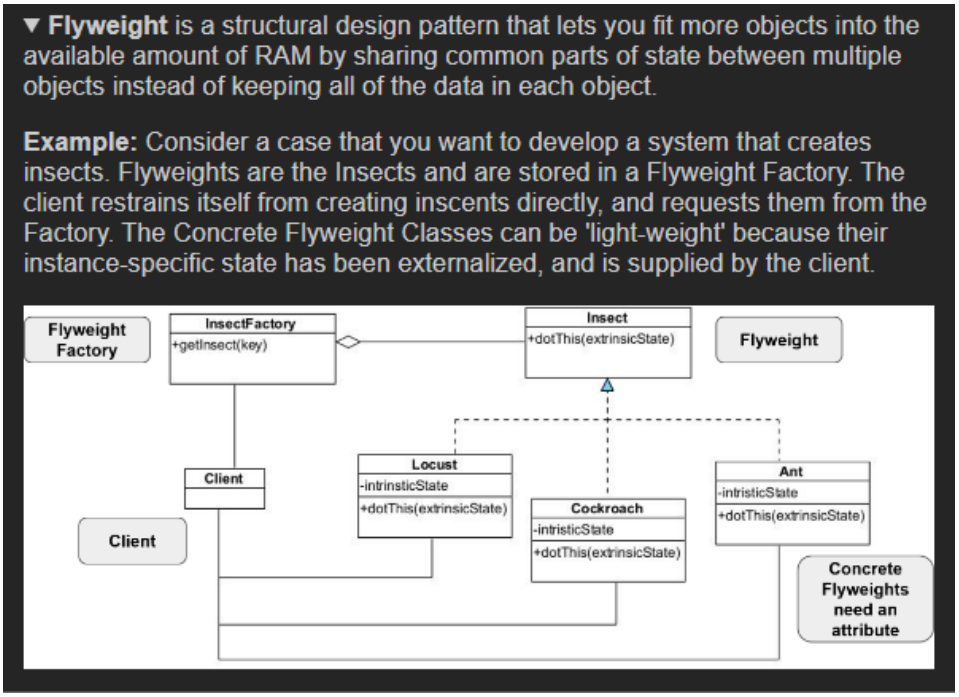
Screenshot 8. The expandable panel of Composite



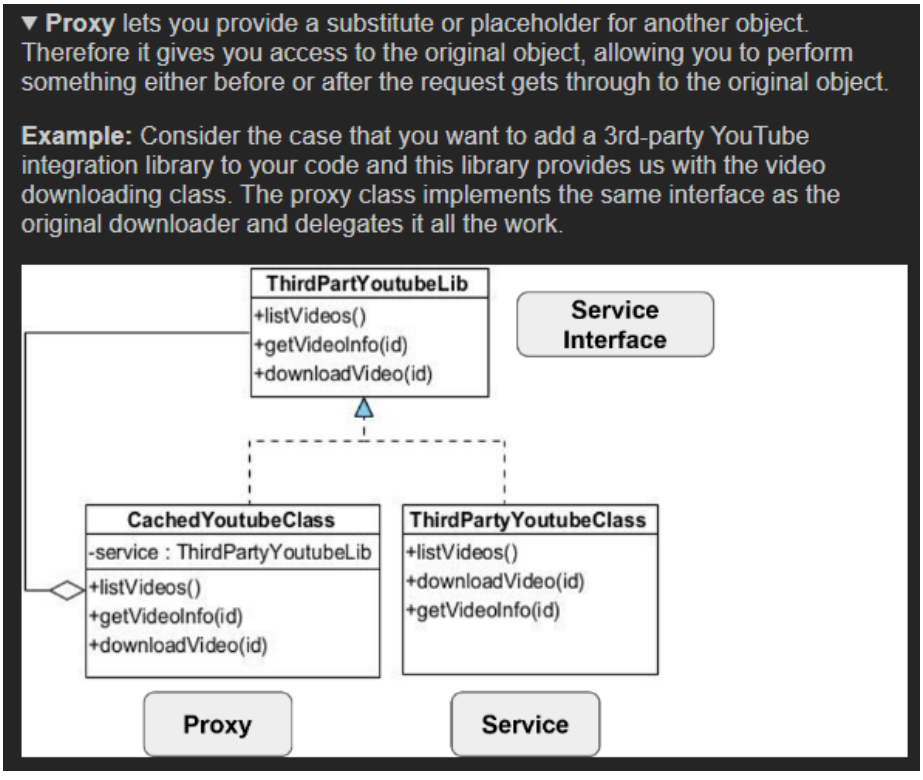
Screenshot 9. The expandable panel of Decorator



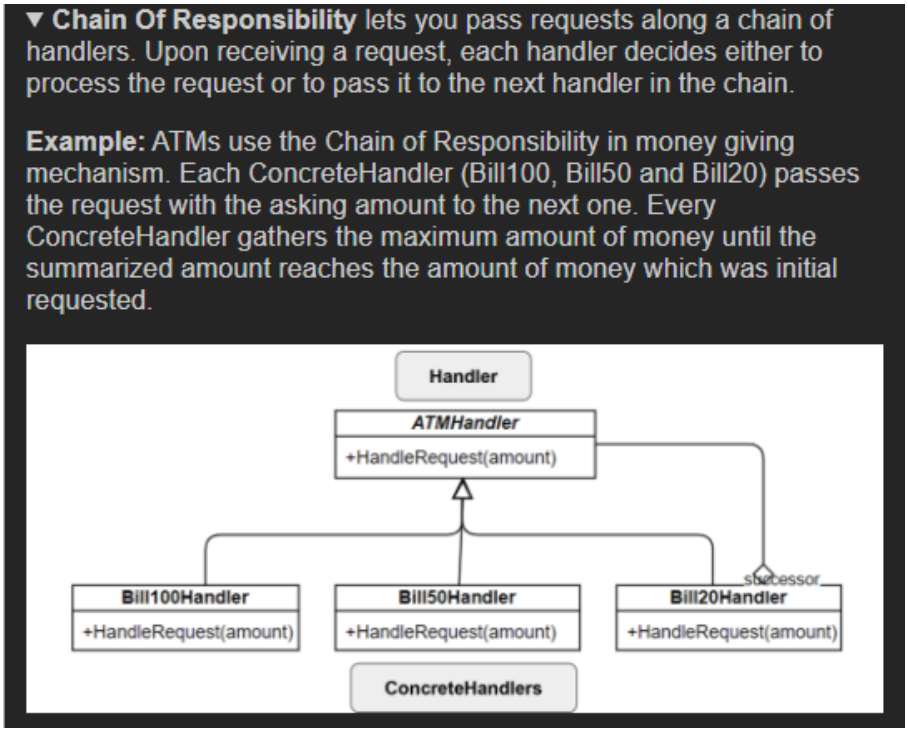
Screenshot 10. The expandable panel of Facade



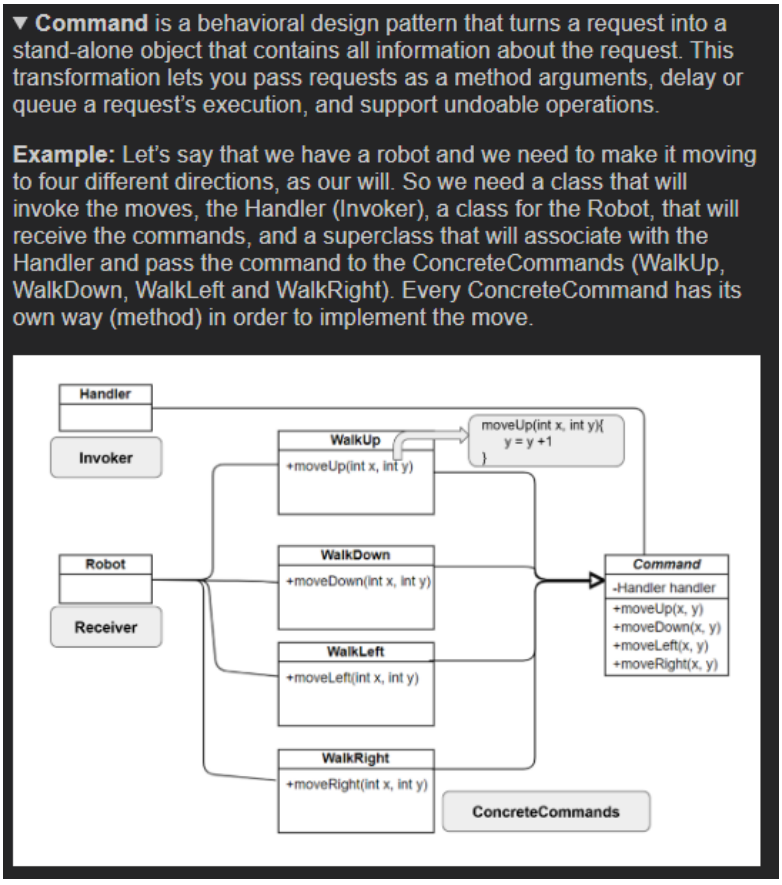
Screenshot 11. The expandable panel of Flyweight



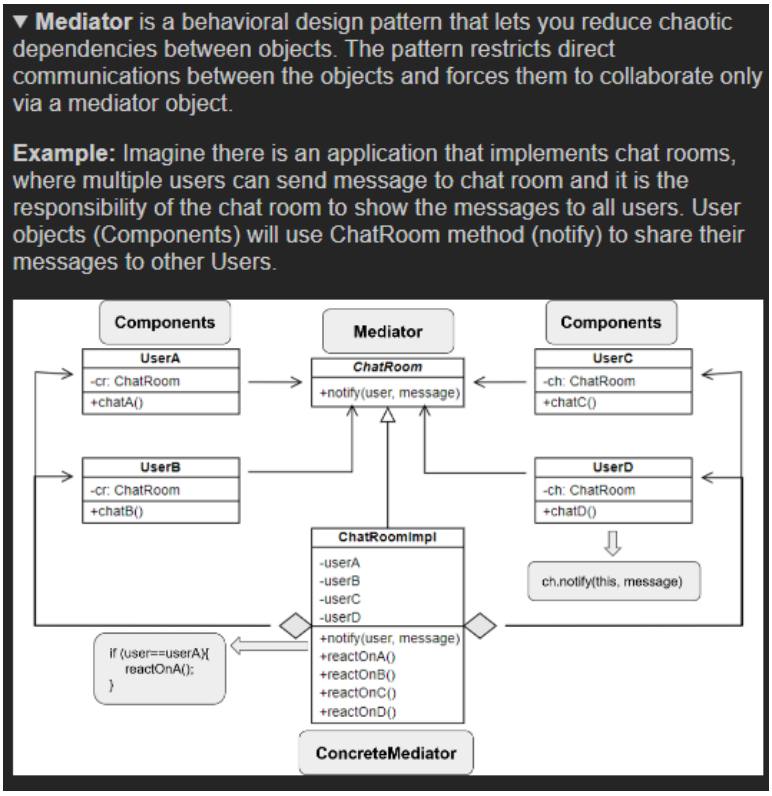
Screenshot 12. The expandable panel of Proxy



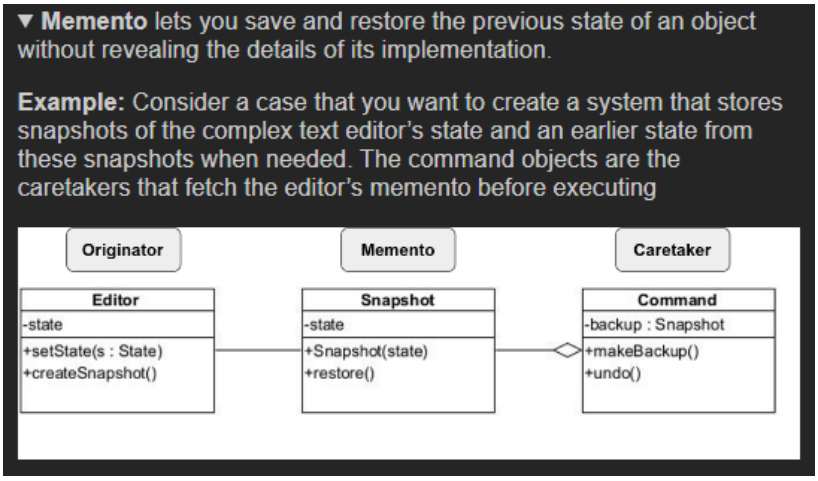
Screenshot 13. The expandable panel of Chain Of Responsibility



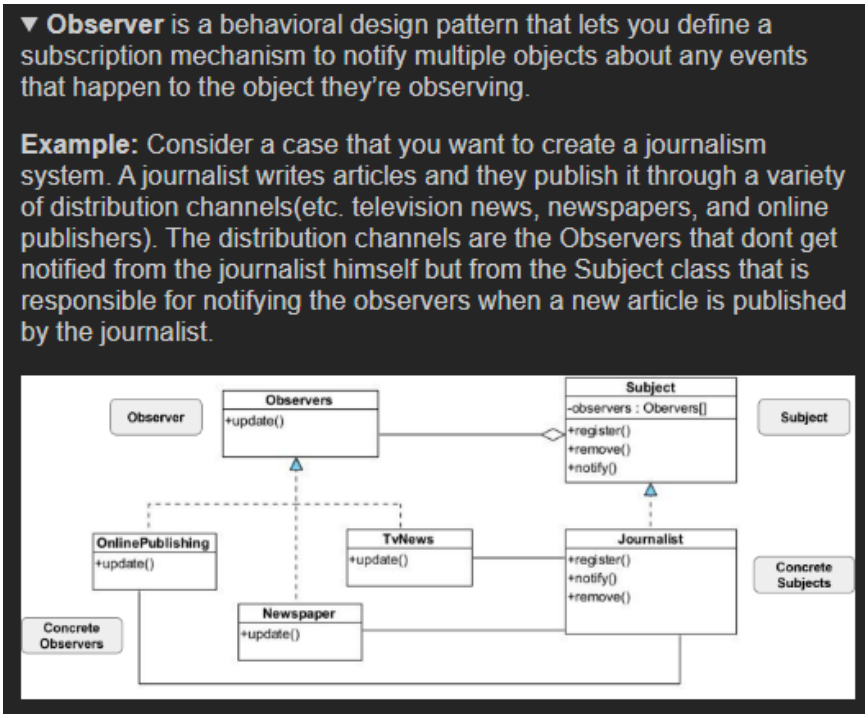
Screenshot 14. The expandable panel of Command



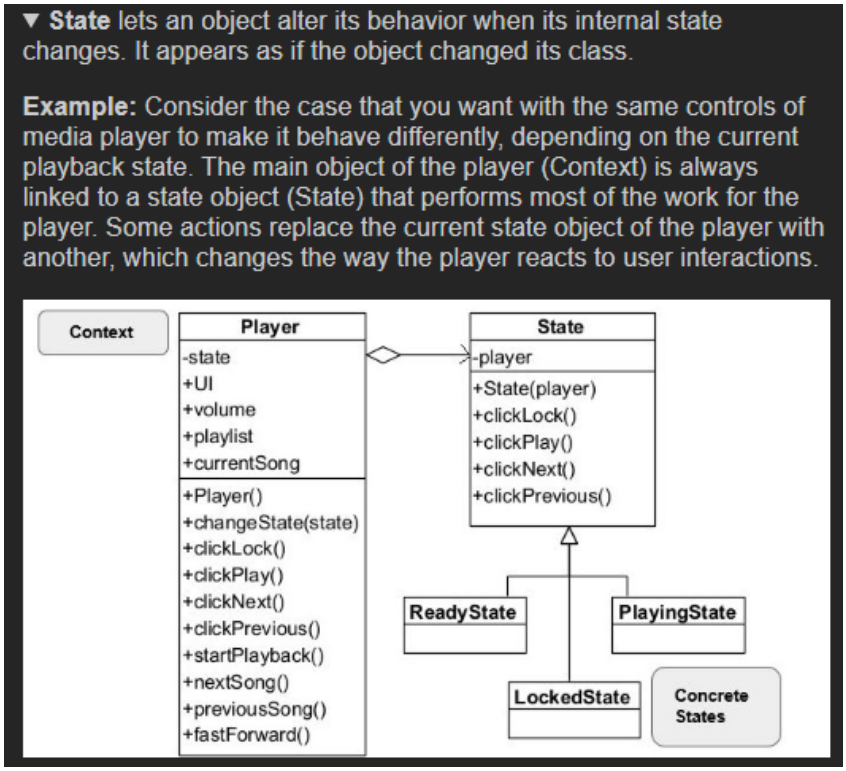
Screenshot 15. The expandable panel of Mediator



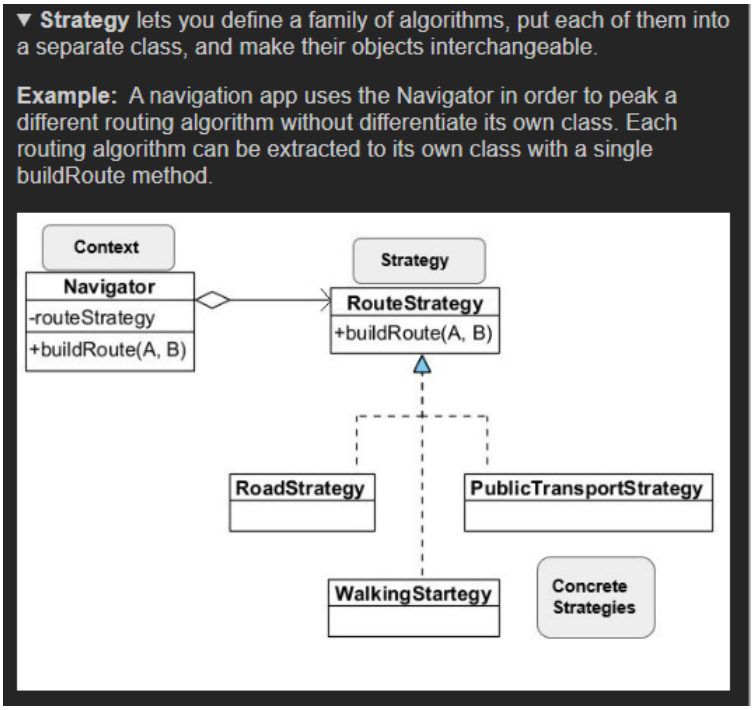
Screenshot 16. The expandable panel of Memento



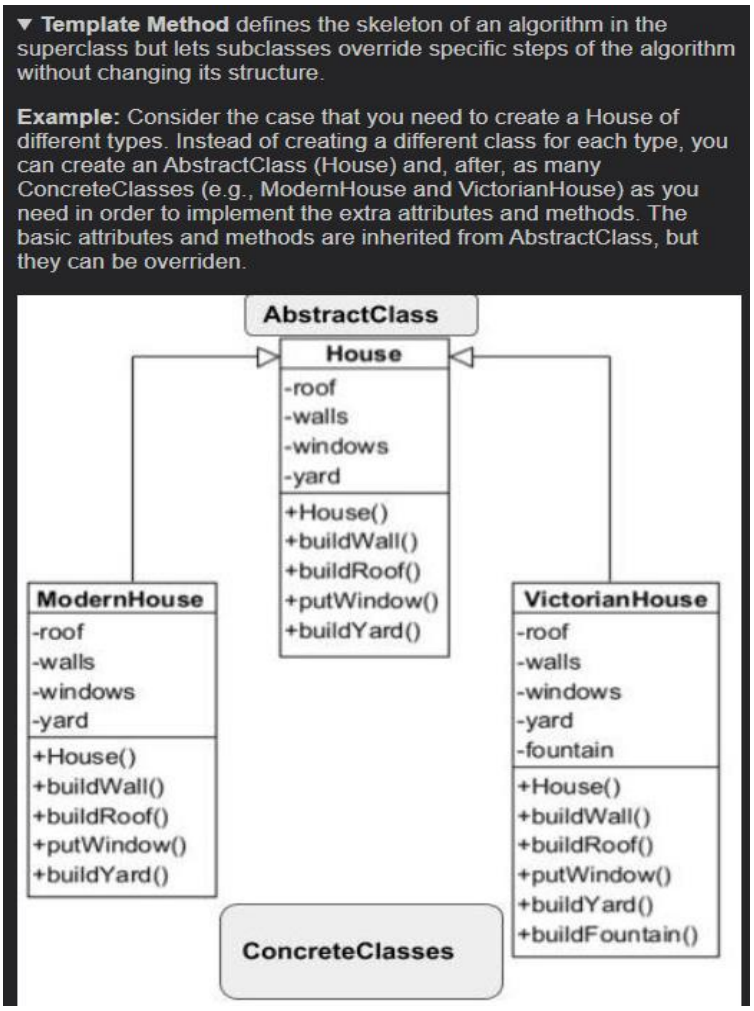
Screenshot 17. The expandable panel of Observer



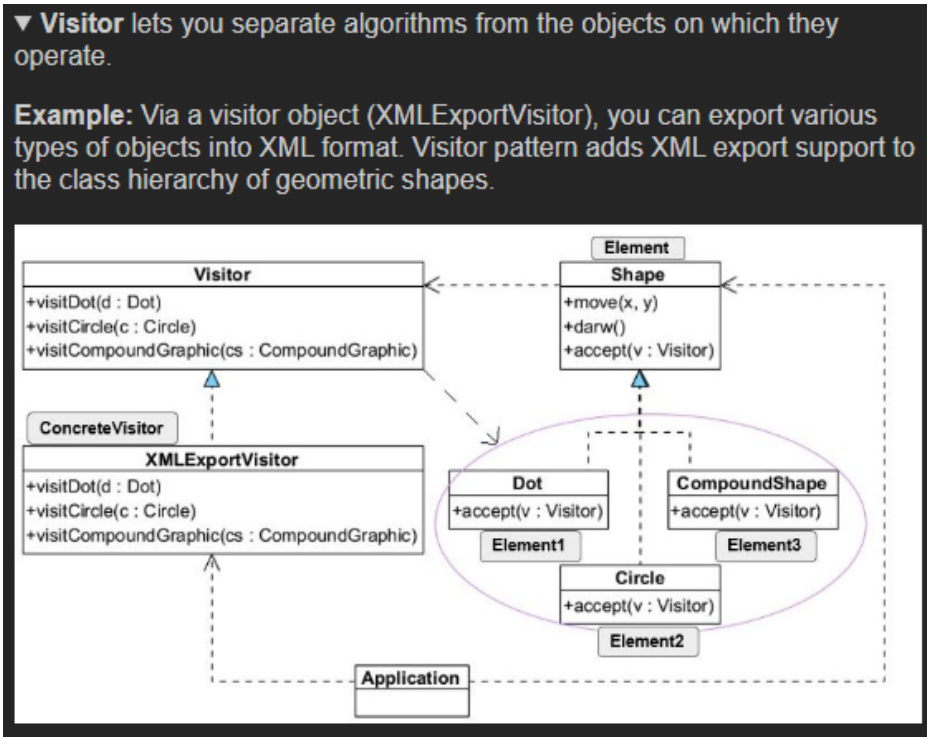
Screenshot 18. The expandable panel of State



Screenshot 19. The expandable panel of Strategy



Screenshot 20. The expandable panel of Template Method



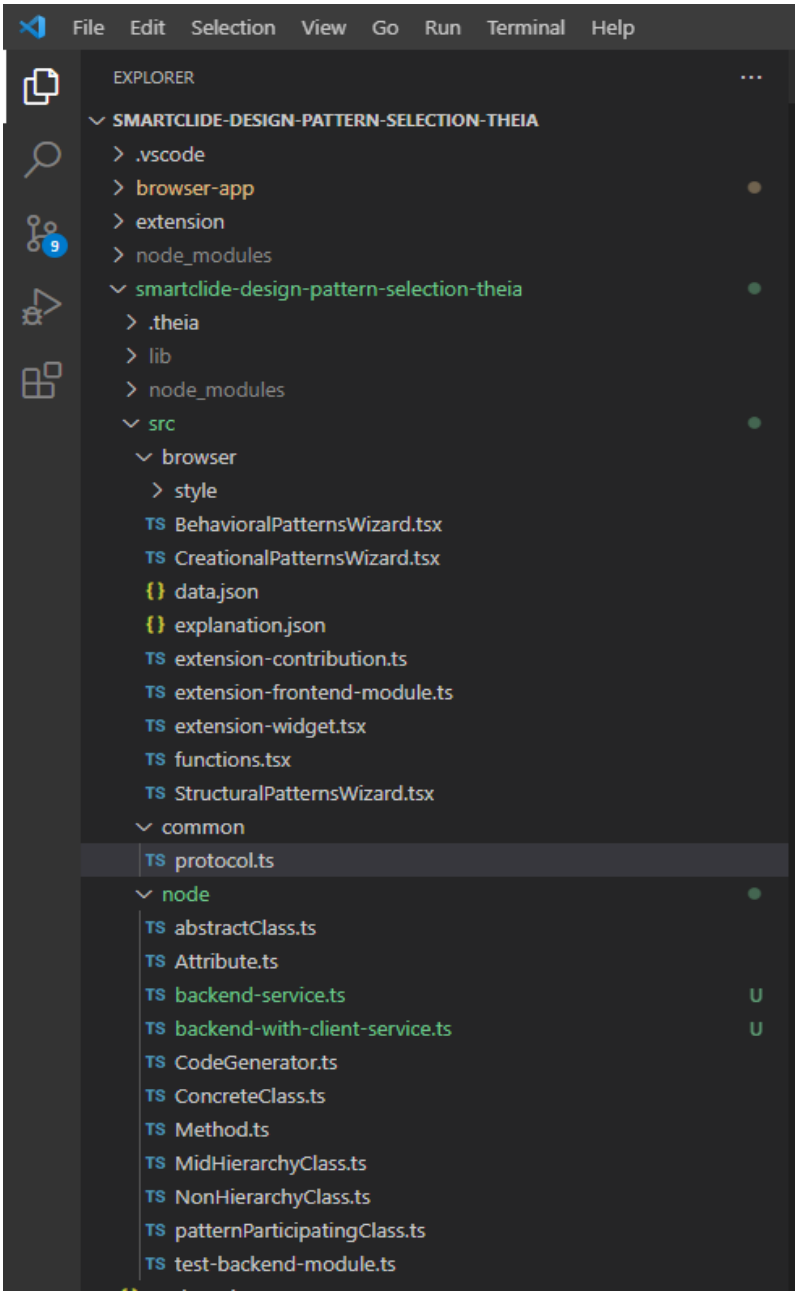
Screenshot 21. The expandable panel of Visitor

3.2.3 Tool Architecture

The "src" of the application consists of two basic directories: the "browser" directory and the "node" one. Also, there is a third directory, the "common" directory, which, basically, connects the other two directories. The "browser" directory contains all the files that refer to the Front-end, while the "node" directory contains the files that make up the Back-end.

The connection link between them is the `protocol.ts` file, which can be found in the "common" directory. The `protocol.ts` file is presented in Code Snippet 1. As you can see, there is an interface for Back-end services that are being exported. These services are the methods: `getMethods`, `getFileNames` and `codeGeneration` and they will be presented at some point below of this paragraph.

Code Snippet 1: File explorer of the project.



Code Snippet 2: The `protocol.ts` file connects the Front-end with Back-end

```
import { JsonRpcServer } from '@theia/core/lib/common/messaging';

export const HelloBackendService = Symbol('HelloBackendService');
export const HELLO_BACKEND_PATH = '/services/helloBackend';

export interface HelloBackendService {
    getMethods(getUrl: string, fileName:string): Promise<string[]>;
    sayHelloTo(url: string): Promise<string[]>;
    codeGeneration(cName : string, jsonObj: string,
```

```
statePatternSelection: string): Promise<string>;

}
export const HelloBackendWithClientService =
Symbol('BackendWithClient');
export const HELLO_BACKEND_WITH_CLIENT_PATH = '/services/withClient';

export interface HelloBackendWithClientService extends
JsonRpcServer<BackendClient> {
    greet(): Promise<string>
}
export const BackendClient = Symbol('BackendClient');
export interface BackendClient {
    getName(): Promise<string>;
}
```

The Front-end of the application is composed of two parts. The first one implements the first option that is being given to the user, more specifically the selection of a pattern from the drop-down menu when he/she feels confident with the selection of the pattern that will be used(***EXPERT-MODE***). The second part is responsible for implementing the other user option, the use of the WIZARD(***WIZARD-MODE***). Both parts use a JSON object literal that is stored in the `data.json` file. The JSON Object's literal keys are the names of the GoF Design Patterns, excluding Iterator and Interpreter Design Pattern.
The values of these keys are objects that have one key called "values". The "values" is an object that illustrates the structure of each Design Pattern. The structure object has its own key/value pairs, where the keys are named after the role names of a specific Pattern. The available types of roles are the following: Class, Method, Attribute, and method Parameter. The key name of each role helps the user to distinguish the role type, consequently the key name contains the name of the type, except for the case that the role is a class. Furthemore, the value of the key/pair of the structure object is an object that contains two key/value pairs.The first key is called "name" and its value is going to take the name of the specific role from the user. The second one is called "extension" and its values are 0 or 1 (**0**:indicates that the user can not add another role of this type, **1**: the user can add new roles of this type).

Code Snippet 3: Structure of the Command Pattern in the JSON file

```
    "Command":{
        "values":{
            "Receiver":{
                "name":"",
                "extension":0
            },
            "Invoker":{
                "name":"",
                "extension":0
            },
            "Command":{
```

```
                "name":"",
                "extension":0
            },
            "ConcreteCommand1":{
                "name":"",
                "extension":1
            },
            "ConcreteCommand1Method":{
                "name":"",
                "extension":0
            },
            "ConcreteCommand1MethodParameter1":{
                "name":"",
                "extension":1
            }
        }
    },
```

Considering that the user chooses the ***EXPERT-ZONE*** and picks one pattern from the dropdown list, an expandable panel and the structure of the pattern are being displayed. The expandable panel holds information about the pattern that the user chose from the list, such as a mini description of the pattern, an example, and a class diagram. A new JSON Object literal, that is stored in the `explanation.json` file, stores information about each one pattern (Code Snippet 4). This JSON Object literal has 21 key/value pairs for GoF Design Patterns. The key is the name of the Design Pattern and the value of the pair is an object that has two key/value pairs. The first one is for the description of the pattern and the second one is for the example.

Code Snippet 4: Structure of the Adapter and Bridge Pattern in the JSON file

```
"Adapter": {
  "description": "allows objects with incompatible interfaces to collaborate. The adapter implements the interface of one object and wraps the other one.",
  "example": " Consider the case that you have a SquarePeg and you need to fit it in a RoundHole. You need to create an adapter, which receives calls from the client (RoundHole) via the adapter interface (RoundPeg) and translates them into calls to the wrapped service object (SquarePeg) in a format it can understand"
},
"Bridge": {
  "description": "lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.",
   "example": " Consider a case that you need to remotely control more than one device. Instead of creating a different class for controlling every device, you can create a bridge between Remote (Abstraction) and Device (Implementation), which allows you to hande many devices by the Device interface."
},
```

Apart from the expandable panel, the structure of the pattern is a dimensional HTML table that displays the roles of the pattern. Each row represents one role/key of the chosen pattern's structure from the JSON Object literal (data.json) and has two main columns and a third one that is not displayed in every role. The first column is for the label of the role (e.g ConcreteProduct1 and ConcreteFlyweight1Attribute), the following column holds the textfield, where the user is going to type the name that he/she wants to give to the specific role. The values, inserted by the user, in every textfield are checked by certain controls. This operation is implemented in the `checkInputsOnSubmit` method. As presented in Code Snippet 5, the method has a number parameter, the number of the form that is used depending on the selected mode (0:***EXPERT-ZONE***, 1:***WIZARD-MODE***). The `checkInputsOnSubmit` method works with the, just mentioned, form HTML Element, which collects all the inserted values, and checks them in three ways: if there is any empty value, if there is a duplicated one and if the value follows the writing rules of the code entities (Class, Method, Attribute and method Parameter).

Code Snippet 5. The checkInputsOnSubmit checks the inserted values in textfields

```
checkInputsOnSubmit(aaform: number) {
        if (this.checkEmptyInputs(document.forms[aaform] as HTMLFormElement)) {
            return "You need to fill all the fields!";
        } else {
              for (let i = 0; i < (document.forms[aaform] as HTMLFormElement).length; i++) {
                let field = document.forms[aaform][i] as HTMLInputElement;
                if (field.id.includes('txtbox')) {
                    if (this.checkInputsForSameValues(field.value, document.forms[aaform] as
HTMLFormElement)) {
                        return "There are duplicated names in the fields!";
                    }
                }
            }
            for (let i = 0; i < (document.forms[aaform] as HTMLFormElement).length; i++) {
                let field = document.forms[aaform][i] as HTMLInputElement;
                if (field.id.includes('txtbox')) {
                    if (field.id.includes('Attribute') || field.id.includes('Parameter') ||
field.name.includes('Attribute') || field.name.includes('Parameter')) {
                        if (!this.checkAttributeNameWriting(field.value)) return
"Attribute/Parameter's type can start with uppercase letter! Attribute/Parameter's name must
start with small letter! ";
                    } else if (field.id.includes('Method') || field.name.includes('Method') ||
(field.id.includes('Step') || field.name.includes('Step'))) {
                        if (!this.checkMethodNameWriting(field.value)) return "Method's name
must follow camel writing!";
                    } else if (!this.checkClassNameWriting(field.value)) {
                        return "Class's name must start with a capital letter!";
                    }
                }
            }
        }
        return "Input is valid";
    }
```

The controls are carried out as it is shown at Code Snippet 6, 7 and 8, respectively.

Code Snippet 6. The method checkEmptyInputs checks if there is an empty textfield.

```
checkEmptyInputs(vform: HTMLFormElement) {
        for (var i = 0; i < vform.length; i++) {
            if ((vform[i] as HTMLInputElement).value.trim() === "" && !(vform[i] as
        HTMLInputElement).id.includes('btn') && !(vform[i] as
        HTMLInputElement).id.includes('button') && !(vform[i] as
        HTMLInputElement).id.includes('radio')) {
                console.log('TRUE', (vform[i] as HTMLInputElement).id);
                return true;
            }
        }
        return false;
    }
```

Code Snippet 7. The method checkInputsForSameValues checks if there is a duplicated value in textfields.

```
    checkInputsForSameValues(value: string, vform: HTMLFormElement) {
        let count = 0;
        for (let i = 0; i < vform.length; i++) {
            if (value === (vform[i] as HTMLInputElement).value) {
                count++;
                if (count == 2) {
                    return true;

                }
            }
        }
        return false;
    }
```

Code Snippet 8. The methods that check the writing of each programming entity.

```
    checkClassNameWriting(value: string) {
        return (value.match("^([A-Z]{1}[a-zA-Z]*[0-9]*)$")) ? true : false;//class case
    }
    checkMethodNameWriting(value: string) {
        return (value.match("^([a-z]+[a-z|0-9]*([A-Z][a-z|0-9]*)*)$")) ? true : false;//method
        case
    }
   checkAttributeNameWriting(value: string) {
      return (value.match("^([A-Za-z][a-z]+( [a-z][a-zA-Z0-9]*))$")) ? true : false; //attribute
        case
    }
```

The insertion of the rows is the responsibility of the `insertCells` method (Code Snippet 9). Each row is inserted in an alphabetical order (depending on the label). The `insertCells`

method calls the method that creates the components of type label (`createLabel`) and the method that creates the components of type input (`createInput`). Except for the two main columns that are being displayed for each role, another column is added when the extension key of the specific role has the value **1.** The creation of the button (plus button) is the responsibility of the `createButton` method. Moreover, by clicking the plus button, new rows are added to the table, depending on the pattern that the user has chosen. In some patterns, by raising the button click event, multiple roles are added. For example, in the Factory Method, if we want to add another type of the role "ConcreteCreator1" by clicking the plus button, two new rows are going to be added. The first new row is for the ConcreteCreator2 and the second one is for the ConcreteProduct2 because each one ConcreateCreator is linked with a ConcreteProduct (for more information about the pattern see paragraph 2.1.1).

Code Snippet 9: The insertCells method.

```
insertCells(table: HTMLTableElement, key: string,) {
        if (extensionWidget.functions.check(key,
     extensionWidget.state.statePatternSelection)) {
            let index = 0;
            for (var i = 0; i < table.rows.length; i++) {
                let label = (document.getElementById('label' + (i + 1)) as
     HTMLLabelElement).innerHTML;
                if (key.localeCompare(label, undefined, { numeric: true, sensitivity: 'base'
     }) > 0) {
                    index++;
                }
            }
            let row = table.insertRow(index);
            let cell1 = row.insertCell(0);
            let cell2 = row.insertCell(1);
            cell2.id = "cell2";

            extensionWidget.functions.createLabel(key, "label" + table.rows.length, cell1);
            extensionWidget.functions.createInput(key, "txtbox" + table.rows.length, "",
     "txtbox" + table.rows.length + key, "text", cell2)

     if
     (extensionWidget.data[extensionWidget.state.statePatternSelection].values[key].extensi
     on == 1) {
                let cell3 = row.insertCell(2);
                extensionWidget.functions.createButton("+", "btn" + key, table)
                cell3.appendChild(document.getElementById("btn" + key) as HTMLButtonElement);
                (document.getElementById("btn" + key) as
     HTMLButtonElement).addEventListener('click', (event) => {
                    this.extensionButtonClick(table, (event.target as Element).id,
     extensionWidget.data[extensionWidget.state.statePatternSelection].values);
                });
            }
        }
```

```
}
```

Furthermore, it is important to mention that the inputs have an autocomplete widget and when the user types something the list with the suggestions of the files that already exist are shown. This happens for two reasons. Firstly, by the autocomplete list that appears to the user, the user gets informed about the names of the existing classes, in order to avoid classes with duplicated names. Secondly, the user may want to use an already existing class and in some cases like the Adapter pattern, the user has to give an input that already exists. For this feature, we used the `Autocompleter` library from npm, as shown below.

Code Snippet 10: The showSuggestions method.

```
showSuggestions(value: string, table: string[], id: string, parent: HTMLDivElement) {

        var items = table.map(function (n) { return { label: n } });

        autocomplete({
            input: document.getElementById('txtbox' + id.substring(6,)) as HTMLInputElement,
            minLength: 1,
            onSelect: function (item: AutocompleteItem, inputfield: HTMLInputElement) {
                inputfield.value = item.label!;
            },
            fetch: function (text, callback) {
                var match = text;
                let reg = new RegExp('^' + match, 'i');
                if (match != "") {
                    callback(items.filter(function (n) {
                        if (n.label.match(reg)) {
                            return n;
                        }
                    }));
                }
            },
            render: function (item, value) {
                var itemElement = document.createElement("div");
                itemElement.className = "suggestions";
                var regex = new RegExp('^' + value);
                var inner = item.label!.replace(regex, function (match) { return match });
                itemElement.innerHTML = inner;
                return itemElement;
            },
            customize: function (input, inputRect, container, maxHeight) {
                container.style.visibility = 'visible';
                container.style.left = "auto";
                container.style.top = "auto";
                container.style.position = 'absolute';
                container.style.maxHeight = "140px";
```

```
                container.style.width = "166.400px";
                container.style.background = '#3c3c3c';
                parent.appendChild(container);
            },
            showOnFocus: true,
            disableAutoSelect: true,
        })
    }
```

The first connection in ***EXPERT-MODE*** between Front-end and Back-end is in the `runprocess` method, whereas the Front-end requests the list that contains the existing files' name of the project that the user has opened. This list is the returning promise of the asynchronous `getFileNames` method. Promises are objects in Typescript that are highly used in asynchronous programming. Apart from this, they enable the skipping of the current task and go to the next line of the code[X]. The method's main responsibility is to take the url that is being given as an input and call the backend method `ThroughDirectory`. This method searches recursively through each directory of the url, in order to find a file name. When a file name is found it is pushed in the `Files` list and the full path of the file is added in the `Absolutes` list.

The next connection between Front-end and Back-end is when the user has chosen the Adapter pattern and in this case one of the roles of the pattern is called AdapteeMethod. This role takes as input only methods that already exist in the Adaptee class. By clicking the button "Get Code", the `getCode` method is called, whereas the asynchronous `getMethods` (see Code Snippet 11) from the Back-end is called. The method's main responsibility is to find a list of all the method names of a given file. Firstly, the method finds the absolute path of the `fileName` parameter and then with the use of Regular Expressions returns a promise that contains the method names of the Adaptee class. This list is used in order for checking if the input name that the user typed for the AdapteeMethod is in the list. If the input is incorrect an error message is displayed.

Code Snippet 11: The getMethods method.

```
async getMethods(fileName: string): Promise<string[]>{
        var fs = require("fs");
        let lO = {label: []};
        var res= HelloBackendServiceImpl.absolutes;
        var file=""
        res.forEach(element => {
           if (element.includes(fileName+".java"))
                file = element;
        });
        try {
            const data = fs.readFileSync( file , 'utf8')
            const regex = new RegExp( /(?:(?:public|private|protected|static|final|native|
synchronized|abstract|transient)+\s+)+[$_\w<>\[\]\s]*\s+[\$_\w]+\([^\)]*\)?\s*/gm);
            const array = [...data.matchAll(regex)];
            for(var i = 0; i<array.length; i++){
                var firstString = (array[i].toString()).split('(');//?
```

```
                var secondString = (firstString[0].toString()).split(/\s+/);
                var item = secondString[secondString.length-1];
                this.fillPromise(lO, item);
            }
        } catch (err) {
            console.error(err)
          }
          return new Promise<string[]>(resolve => resolve(lO.label));
    }
```

After the user clicks on the "Get Gode" button of the ***EXPERT-MODE***, given the case that the inputs that were inserted pass all the controls, the inputs are being inserted into the structure of the chosen Design Pattern of the JSON Object Literal (data.json) by the `updateJsonObject` method (Code Snippet 12). In certain cases, when the chosen pattern is one of the followings : Abstract Factory, Factory Method, Builder, expect for the `updateJsonObject` method, the corresponding method is called(`insertInputsAbstractFactory`, `insertInputsBuilder`, `insertInputsFactoryMethod`) in order to fill the key/value pairs, that do not take inputs from the user or their name is compound of two strings. For example, if the chosen pattern is the Builder, the ConcreteBuilders' name consists of the name of the ConcreteProduct and the string "Builder" is appended to the end of the string.

Code Snippet 12: The updateJsonObject method.

```
updateJsonObject(data: string) {
        let values = JSON.parse(JSON.stringify(data));
        let table = document.getElementById('show_pattern_table') as HTMLTableElement;
        for (let i = 0; i < table.rows.length; i++) {
            let label = (document.getElementById('label' + (i + 1)) as
HTMLLabelElement).innerHTML;
            let txtbox = (document.getElementById('txtbox' + (i + 1)) as
HTMLInputElement).value;
            values[label].name = txtbox;
        }
        return values;
    }
```

After the update of the JSON Object Literal, the method `codeGeneration` of the Back-end is called and is responsible for the creation of the new java files. The method takes as parameters the updated JSON Object Literal, the url of the project that the user has opened in the platform and the chosen pattern. Firstly, a new object of the `CodeGenerator` class is created and then, according to the pattern, the corresponding method of the `CodeGenerator` is called(e.g if the user has selected the Flyweight pattern the `flyweight` method of the `CodeGenerator` is going to be called).

The `CodeGenerator` is a class that contains 21 methods for the 21 design patterns. Each one method fills a list of `PatternParticipatingClass` objects for the role type class of the structure of the JSON Object Literal and returns the list to the `getCode` method.

The `PatternParticipatingClass` is an abstract class that has two attributes: one list that stores `Method` objects and another one with type `Attribute`. The classes that extend this class are the following: `AbstractClass`, `NonHierarchyClass`, `ConcreteClass`. These classes implement the abstract method, named `writeToFile`, which is responsible for creating new java files or appending data in the existing file of the project. Furthemore, the `PatternParticipatingClass` class implements the `writeMethods` and `writeAttributes`, that are responsible for writing the methods and the attributes in the file.

For the other role type in the `CodeGenerator` method, a new object of the class `Method` or the class `Attribute` are being created respectively, and then it is passed as an parameter in the `addMethod` or `addAttribute` method of the `PatternParticipatingClass`. Then, after the creation of the list of the classes for each one object of the list, the method `writeToFile` is called.

Code Snippet 13. The method, called from CodeGenerator method, for State pattern

```
public state(jsonObj: string): Array<patternParticipatingClass> {
      let ppc: Object = { object: [] }
      let obj = JSON.parse(JSON.stringify(jsonObj));
      let file1: patternParticipatingClass = new NonHierarchyClass(obj.Context.name);
     file1.addAttribute(new     Attribute(obj.State.name.toLowerCase(),     obj.State.name,
"private"));

        file1.addMethod(new Method(obj.Context.name, "", false, "public", "\t \t this." +
obj.State.name.toLowerCase()  +  "  =  "  +  obj.State.name.toLowerCase()  +  ";",  [new
Attribute(obj.State.name.toLowerCase(), obj.State.name, "")]));
        file1.addMethod(new Method("changeState", "void", false, "public", "\t \t this." +
obj.State.name.toLowerCase()  +  "  =  "  +  obj.State.name.toLowerCase()  +  ";",  [new
Attribute(obj.State.name.toLowerCase(), obj.State.name, "")]));
        let file2: patternParticipatingClass = new abstractClass(obj.State.name);

        Object.keys(obj).forEach((key) => {
            if (key.includes("ConcreteState")) {
                let  file3:  patternParticipatingClass  =  new  ConcreteClass(obj[key].name,
obj.State.name);
                file3.addAttribute(new             Attribute(obj.Context.name.toLowerCase(),
obj.Context.name, "private"));
                file3.addMethod(new  Method("setContext",  "void",  false,  "public",  "\t  \t
this." + obj.Context.name.toLowerCase() + " = " + obj.Context.name.toLowerCase() + ";", [new
Attribute(obj.Context.name.toLowerCase(), obj.Context.name, "")]));
                this.fillPromise(ppc, file3);
            }
        });
        this.fillPromise(ppc, file1);
        this.fillPromise(ppc, file2);

        return ppc.object;
```

```
}
```

WIZARD-MODE architecture begins with the method: `runWizard` in the `extension-widget.tsx` file. This method is called when the user clicks the "Wizard" button and is responsible for creating the UI components of the ***WIZARD-MODE.*** The `runWizard` consists basically of HTML Elements and DOM operations. It is responsible for calling the right class and method, depending on the asked group of patterns of the user. For example, if the user chooses Structural patterns, then the `runWizard` method must call `structuralPatternsWizard` method in the `StructuralPatternsWizard.tsx` file. Keeping Structural Patterns as an example for continuing the explanation of the architecture, a few more things need to be mentioned. First of all, the main thinking behind the UI is the nested Event Listeners, in order for the user to navigate through the questions of the Decision Tree (see paragraph 3.1). Secondly, the methods that generate the HTML Components are common with the ***EXPERT-MODE***: `createLabel, createInput` and `createButton`. But, beside these, there has been some extra categorization for the creation of the needed components through methods: `radioQuestion` (for the binary questions of the Decision Tree) and `textfieldQuestion` (for the other questions). Lastly, when the user clicks the "Get Code" button of ***WIZARD-MODE***, the most internal code of the code nest is executed, in order for the JSON structure to get filled with the Class names, Method names etc. that the user inserted. If we consider that the path of the Decision Tree ended up to the Composite pattern, then the executed code will be that in Code Snippet 14.

Code Snippet 14. The part of the code that fills the JSON structure with theClass names, Method names etc. that the user inserted for Composite pattern.

```
buttonCodeCP.addEventListener('click', async (e: Event) =>{
    let infoList = document.getElementsByClassName('infoField') as HTMLCollection;

StructuralPatterns.values["Composite"].values["Component"].name =
(infoList.item(0) as HTMLInputElement).value;
let numInterfaceMethods =
parseInt((document.getElementById('NumOfInterfaceMethods1') as
HTMLInputElement).value);
let numSimpleObj = parseInt((document.getElementById('NumOfSimpleObjectsTypes1')
as HTMLInputElement).value);

for (var i = 1; i <= numInterfaceMethods; i++) {
StructuralPatterns.values["Composite"].values["ComponentMethod" + i] = { "name":
"", "extension": 1 };
let v1 = (infoList.item(i) as HTMLInputElement).value;
StructuralPatterns.values["Composite"].values["ComponentMethod" + i].name = v1;
}

for (var j = 1; j <= numSimpleObj; j++) {
StructuralPatterns.values["Composite"].values["ConcreteComponent" + j] = { "name":
"", "extension": 1 };
let v1 = (infoList.item(i) as HTMLInputElement).value;
StructuralPatterns.values["Composite"].values["ConcreteComponent" + j].name = v1;
i++;
}
```

```
StructuralPatterns.values["Composite"].values["Composite"].name =
(infoList.item(i) as HTMLInputElement).value;
let message = StructuralPatterns.functions.checkInputsOnSubmit(1);

if (message == "Input is valid"){
StructuralPatterns.functions.checkMessage(await
helloBackendService.codeGeneration(window.location.href,
StructuralPatterns.values["Composite"].values, "Composite"), messageService);
}else{
messageService.info(message);
}
});
```

As we see on the above Code Snippet, finally, the method `codeGeneration` is called in order for the generation of the pattern's classes. The procedure of the Code Generation has been explained above.

4. Industrial Validation

To evaluate the proposed solution, we have performed an industrial validation with a mixed set of novice and experienced software engineers. In this section we present the industrial validation study protocol, based on the guidelines of Runeson et al. (2012). In Section 4.1, we set the objectives and research questions, in Section 4.2 the study setup, whereas in Section 4.3 we present the data collection and analysis approaches to ensure data triangulation and answer the research questions.

4.1 Objectives & Research Questions

The main goal of the SmartCLIDE platform is to be relevant to the software industry (i.e., advance the state-of-practice in pattern selection), to be usable, and aid the correct and timely pattern selection. According to the aforementioned goals we have derived three research questions (RQ):

RQ_1: Is the proposed pattern selection approach industrially relevant?

The first step in ensuring the industrial relevance of a research prototype is the investigation of the current industrial practices. Before performing the evaluation of the proposed approach and tool, we first need to understand the current way in which patterns are selected. Next, we can understand and assess if the proposed approach and tool treat existing limitations and retain the strong points. The benefits and drawbacks of the SmartCLIDE pattern selection approach will be the main outcomes of answering this research question.

RQ_2: What is the effectiveness of the proposed approach in terms of pattern selection?

This research question will focus on the effectiveness of the proposed approach in terms of correctness and timeliness. In particular, we explore if the participants are aided in selecting the intended pattern, in each mode of the Theia Extension (i.e., `EXPERT-MODE` and `WIZARD-MODE`), as well as the time required to complete the tasks. Apart from the quantitative analysis, a qualitative assessment on how helpful in terms of correctness and timeliness each feature (EXPERT-MODE, WIZARD-MODE, CODE-GENERATION) is, has been discussed with the practitioners. An important parameter in answering this question is the level of expertise of the software engineer (novice / experienced).

RQ_3: What is the usability of the accompanying tool?

Apart from being relevant and useful in practice, in order for a research prototype to be industry-ready, a key factor is to be usable. Through this research question, we focus on the usability of the Theia Extension, assessing its ease of use, learning curve etc. The outcome of this research question is of paramount importance to the Research & Development team of the SmartCLIDE project for improvement suggestions, as well as the interested practitioners, since it guarantees to some extent the end-users' experience.

4.2 Industrial Study Setup

To answer the aforementioned questions, we have performed an embedded single-case study in the software industry (Runeson et al., 2012). The case of the study is a European software development company (at the SME level) with Headquarters in Germany (Cologne) and a branch in Greece (Thessaloniki), namely Onelity. Onelity offers full custom service or turnkey package solutions on IT projects. The study is embedded, in the sense that inside the single case, more than one unit of analysis have been studied. The units of analysis correspond to the 15 participants (software engineers and lead software engineers) of the case study. Some demographics of the participants are presented in Table 1 (the experience is measured in years).

***Table 1**. Study Demographics*

	1-2 years	**3-6 years**	**7+ years**
Working Experience	6	6	3
	Almost None	**Some**	**Experts**
Patterns Experience	10	3	2

The study was conducted as a half-day workshop, held at the premises of Onelity. The workshop was organized as follows:

[Part A] Pre-study questionnaire (10 minutes).

[Part B] A short presentation of how the Theia Extension works, so as for the participants to get familiar with the tool (20 minutes).

[Part C] The participants will be assigned a first task, using the ***EXPERT-MODE*** of the Theia Extension (30 minutes)

[Part D] The participants will be assigned a second task, using the ***WIZARD-MODE*** of the Theia Extension (30 minutes)

[Part E] A focus group was performed with the participants so that a qualitative assessment to be reached (90 minutes).

[Part F] Post-study questionnaire (10 minutes)

The focus group duration was intentionally made quite long, so that a long range of topics to be discussed, and enough time has been given to all participants to make their positioning. In Table 2, we present the task distribution to participants (Parts B and C). The participants are anonymous and are referred to as P1-P15. The distribution of the participants was random, but some constraints were applied: (a) that one participant must take one task in the ***EXPERT-MODE*** and one in the ***WIZARD-MODE***; and (b) the tasks for the same pattern cannot be assigned to the same participant. We note that the tasks are named after the intended pattern to be used (but this information was hidden from the participants of the industrial study). The tasks and details on the data collection instruments are provided in Section 4.3.

***Table 2**. Participants Assignment to Tasks*

Participant ID	**Task for EXPERT-MODE**	**Task for WIZARD-MODE**
P1	Factory Method	Observer
P2	Builder	Strategy

P3	Strategy	Memento
P4	Memento	Command
P5	Command	Factory Method
P6	Bridge	Observer
P7	Composite	Builder
P8	Bridge	Composite
P9	Factory Method	Memento
P10	Memento	Builder
P11	Composite	Bridge
P12	Strategy	Bridge
P13	Builder	Strategy
P14	Bridge	Command
P15	Command	Factory Method

4.3 Data Collection & Analysis

Data Collection: We collected data through different collection methods, as presented in Table 3 and discussed below. For all research questions, method triangulation has been applied to increase the validity of the findings. Method triangulation refers to the technique of mixing more than one method to gather data (e.g., task analysis, questionnaires, and a focus group) to answer a research question, so as to reduce bias, and raise confidence in the results.

Table 3. *Data Collection Methods per Research Question*

Collection Method	**RQ_1**	**RQ_2**	**RQ_3**
Focus Group	X	X	X
Questionnaire	X		X
Task Analysis		X	

Regarding RQ_1, we have worked on the data gathered from the focus group. The goal of RQ1 was to understand the ***state-of-practice*** in the company for ***pattern selection***, and identify the ***benefits*** that can be obtained by using the proposed approach. In the focus group we have used four questions related to the answer of RQ_1 (see below). Also, data from the pre-study questionnaire have been used, related to patterns experience and programming experience

What is your experience with DP? How do you choose which DP to use, or if you will use it? Was the approach and tool helpful? What are the perceived main benefits? Which mode would you choose in your work routine if the implementation of a design pattern was needed?

To answer RQ_2, i.e., assess ***effectiveness*** of the approach, as well as the ***mode*** of ***operations*** and ***main features***, we have relied on task analysis and the focus group. As explained in Section 4.2 the participants were given two tasks to work on using both modes of the Theia extension. Some task examples are presented below:

Factory Method Task: Suppose a bank that offers credit cards to their customers. Assuming that they offer 3 types of credit cards, such as Silver, Gold and Platinum and each card has a different credit limit. You are asked to implement a system that creates cards of all possible types (Patra, 2020).

Factory Method Task: Suppose we are in a factory which produces windows in polygonal shapes. There are three window's shapes: triangle, rectangle and pentagon. Your task is to implement a class system for the production of all three window's shapes in the factory.

Builder Task: Imagine the case that you want to develop a system that creates menus for a fast-food canteen. The canteen is famous for their meals because they are at a reasonable price. A typical meal, consists of the main part (beef burger or vegan burger), the bread (brioche or typical bun), the sauce (cheddar sauce, parmesan sauce, or BBQ sauce), the sides (French fries, onion rings or sweet potatoes) and the drink (coca cola, beer or sprite). The customer is free to make any selection of parts within each category. However, the process making is the same. Moreover, the meal must contain something from every category (e.g., you cannot order a burger without sauce or without a drink). After the order is ready, the cashier pushes the order to the cook (SourceMaking, n.d).

Builder Task: Consider the case that you want to create a system that creates complex objects for a pizzeria. A typical pizza, consists the dough (typical dough, dough with philadelphia in the crust, or dough with sausage in the crust), of the sauce(typical tomato sauce, hot sauce or white sauce) and the toppings(cheese + pineapple + ham, cheese + bacon + green peppers or cheese + pepperoni). Note that there can be a variation in order but the process making is the same, whether the customer chooses a crust with sausage or the typical one. In addition, every object must contain something from every category (e.g a pizza can not . After the order is ready, the cashier then gives the order to the cook (SourceMaking, n.d).

Bridge Task: Suppose a software that performs animations of 3D houses' openings. The house openings can be: windows, doors, and roof windows. Each house part can be animated with different sprites (open, close, destroy, change color). For this task you need to create an effective system that limits class combinations and allows the animation of all houses parts, by selecting the proper animation for each possible pair (e.g., open door, open windows, close roof window, etc.).

Bridge Task: Let's say that you need to build a house (or more) that consists of a roof, a wall (or more) and a balcony. Each house must have the same color roof, walls and balcony. Let's say there are three colors (white, red and blue). For this task you need to create an effective

system that limits class combinations and builds houses with the above characteristics (by combining house's parts and colors).

Composite Task: Imagine a cart that sells ice-cream, with two options for toppings: simple toppings and combinations of toppings. Simple toppings can be: chocolate, strawberries, truffle and cookies. Your task is to implement a system through which a calcCost method in the IceCream class can return the cost of any tentative ice cream (handling both simple and complex toppings). For complex toppings the cost is the sum of all simple toppings (Finch, 2020)

Composite Task: Suppose we have a Christmas tree and we want to adorn it. The ornaments can be: topper, tinsel, garland and bubble-lights. The ornaments can also be combined in any possible way. Equip the ChristmasTree class with a render method that is able to draw a tree with simple ornaments or combinations of ornaments (Baeldung, 2022).

Command Task: Imagine a restaurant: The waiter takes an order or command from a customer and encapsulates that order by writing it on the check. The order is then queued for a short order cook. Note that the pad of "checks" used by each waiter is not dependent on the menu, and therefore they can support orders or commands to cook many different dishes. Your task is to create a system where the order from the waiter to the cook contains a dish: fried chicken and a cocktail: martini (which we all know that it needs at least one olive!) (SourceMaking, n.d)

Command Task: Imagine a product line that is automatically handled by robots. Each vehicle consists of three parts (part1, part2, part3). The production line spans across 3 different departments: the first is responsible for producing the parts by using raw materials (material1, material2, material3), the second is responsible for assembling by putting the parts together and the third for the testing by crashing the vehicle in given conditions (rain, dangerous road). You need to create an effective ProductLineController that instructs the robots to perform the necessary tasks.

Memento Task: Suppose we have a painting application where we can paint in layers a painting and by clicking buttons next or previous, we can navigate through the layers of the painting. This painting application can save the state of the painting at a given instant and restore it by clicking the previous button. Your task is to create a system of classes that will implement the function of the previous button for a painting consisting of three different layers.

Memento Task: In soccer, sometimes after the referee awards a penalty or shows a red card, he needs to go and check the VAR. So, there is a chance that the referee is wrong and the state of the game needs to be restored. In this case, there must be a system that can restore the state of the game after a misjudged call by the referee. Your task is to implement such a system where the actions of the game will be able to be restored in a previous state.

Strategy Task: In this task, you need to implement the various payment methods in an e-commerce application. The customer chooses a payment method: either Paypal or Credit Card, after selecting the products they want to use. The checkout form differs for each payment method, and appropriate fields to record the payment details are needed. For Paypal, the customer has to add their email address and their password in the form and then clicks the signed in button. For credit card option, the customer has to enter the credit card number, the day of expiration and the cvv (Refactoring Guru, n.d)

Strategy Task: Imagine that you have a transportation problem where a traveler wants to go to the airport. Several options exist such as driving their own car, taking a taxi, or a city bus. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must choose which means of transportation he/she will select based on factors such as cost, convenience, and time (Refactoring Guru, n.d).

Observer Task: Imagine that we have an application that illustrates the twitter follow button. Twitter is a well-known social networking service and it is highly used by celebrities and famous actors too to interact with their fans, through posts and tweets. If a fan is interested in an actor, they are going to follow them, in order to stay updated and get notified of their tweets. Moreover, the fans will have the opportunity to unfollow celebrities or famous actors when they lose interest. Your task is to create a system of objects where fans can follow a celebrity and get notified of their posts (Andhariya, 2017)

Upon the participants completing the tasks, the researchers have recorded the following variables, so as to serve the quantitative assessment of the proposed approach:

[V1] Task ID

[V2] Theia Extension mode (***`EXPERT-MODE`*** / ***`WIZARD-MODE`***)

[V3] Chosen pattern

[V4] Completion Time

[V5] Correctness in Selection

[V6] Correctness in Code Generation

With respect to the qualitative part of the analysis, the following focus group questions have been considered:

Did you find the examples in the ***EXPERT-MODE*** helpful to understand the patterns?
How confident are you of the choice of the pattern you made (for each task)?
Were the questions of the ***WIZARD-MODE*** clear? Was there any ambiguity?
Was code generation useful?
Was it straightforward to map roles to classes?

Which task did you find more difficult to complete?

Finally, with respect to answering RQ_3, i.e., the ***usability*** of the Theia extension, we relied on three focus group questions, as presented below:

How did you experience the navigability in the tool? Have you encountered any usability issues? What improvements would you suggest for better navigation in the Wizard option?

Whereas from a qualitative point of view, we relied on the SUS questionnaire (Brooke, 1996), which is a state-of-the-art method in the user interface design field. SUS is a reliable tool for measuring usability. It consists of a 10-item questionnaire with five response options for respondents; from Strongly agree to Strongly disagree. Originally created by Brooke (1996), it allows UI/UX experts to evaluate a wide variety of products and services, including hardware, software, mobile devices, websites and applications. The items of evaluation are presented below. The participant's scores for each question are converted to a number, added together and then multiplied by 2.5 to convert the original scores of 0-40 to 0-100. Though the scores are 0-100, these are not percentages and should be considered only in terms of their percentile ranking. Based on the literature, SUS scores above 68 are considered above average and anything below 68 is below average (Brooke, 1996).

I think that I would like to use this system frequently	I thought this system was too inconsistent
I found the system unnecessarily complex	I felt very confident using the system
I thought the system was easy to use	I found the system very cumbersome to use
I think I would need the support of a technical person to be able to use this system	I would imagine that most people would learn to use this system very quickly
I found the various functions in this system were well integrated	I needed to learn a lot of things before I could get going with this system

Data Analysis: To validate the proposed solution, we have used quantitative analysis for providing a synthesized overview of the achieved impacts, and qualitative analysis for interpretation of the results. To synthesize qualitative and quantitative findings, we have relied on the guidelines provided by Seaman (1999). On the one hand, to obtain ***quantitative results***, we employed descriptive statistics and basic hypothesis. For usability, we provided the total SUS score, along with the most common scales for interpretation, in terms of acceptance, adjective, and grade. On the other hand, to obtain the ***qualitative assessments***, we use the focus group data, which we have analyzed based on the Qualitative Content Analysis (QCA) technique (Elo & Kyngas, 2008), which is a research method for the subjective interpretation of the content of text data through the systematic classification process of coding and identifying themes or patterns. This process involved open coding, creating categories, and abstraction. To identify the codes to report, we used the Open-Card Sorting (Spencer, 2009) approach. Initially we transcribed the audio file from the focus group and analyzed it along with the notes we kept during its execution. Then a lexical analysis took place: in particular, we have counted word frequency, and then searched for synonyms and removed irrelevant words. Then

we coded the dataset, i.e., categorized all pieces of text that were relevant to a particular theme of interest, and we grouped together similar codes, creating higher-level categories. The categories were created during the analysis process and were discussed and grouped together through an iterative process in several meetings of both authors and experts. The reporting is performed by using codes (frequency table) and participants' quotes. Based on Seaman (1999) qualitative studies can support quantitative findings by counting the number of units of analysis in which certain keywords occur and then comparing the counts of different keywords, or comparing the set of cases containing the keyword to those that do not.

5.Results

In this section, we present the findings of the empirical evaluation of the SmartCLIDE Pattern Selection approach, organized by research question. Along the discussion features and operation modes are denoted with bold fonts, codes with capital letters, and quotes in italics. In Table 4 we present the codes that have been identified along the discussions of the focus group, accompanied by the most common synonyms, representative quotes and the frequency that participants used them.

Table 4. *Codes of the Qualitative Analysis*

Code	Quote	#
SAVE TIME	"Automating some of the straightforward tasks" "The fact that all patterns are together limits searching time"	13
SOURCE OF KNOWLEDGE	"You can learn about patterns and choose the correct one" "Q&A was helpful since it guides inexperienced developers that lack knowledge to select the right pattern"	8
STAY ON TRACK	"The flow follows the way that a human would think, this helps you stay on track" "I had a pattern in mind from the beginning, but the Q&A did not allow me to go there"	7
DECISION CONFIDENCE	"The Q&A guided me to the solution smoothly, increasing my confidence on my choice" "Although I knew the pattern, the Q&A made me more confident"	6
FITTING FOR NOVICE USERS	"The tool is useful especially for people with low experience in patterns"	5
IMPROVE GUI INTERACTION	"Make the UI more interactive in that part, and enable the selection of the role from the example class diagram, so that the visual information is exploited."	4
MINIMUM REQUIRED CODE	"It is great if we can avoid copying and pasting from internet, which needs to be stripped out of useless parts of the example to add the required business logic"	4
TERMINOLOGY	"The inexperienced developers struggle with the pattern terminology. A tool like that must hide it"	4
LOW LEARNING CURVE	"The tool is very easy to use … I could use it without any guidance	4
PATTERN FAMILIARITY	"Someone needs to first read on patterns, and then use the tool. In that sense, I have a lot of reading to do, before using it efficiently"	3

Code	Quote	#
CORRECTNESS	"The mapping of roles to classes can guarantee the preservation of the pattern rules in the final implementation. It can help in avoiding errors and place the pattern wrongly"	3
CODE READABILITY	"Code generation can also guide in terms of styling, to impose good readability practices, apart from the maintainability benefits"	2
ISOLATING DESIGN FROM CODING	"It is good that the solution links design decisions with code. The fact that code generation is an integral part of the process does not isolate the two and allows to do both from the same environment"	2
CONSISTENCY	"…the theme is consistent to the general layout of Theia…"	1
EXPERIMENTATION	"The tool is also great for experimentation. You can try as many solutions as you wish, check the code and select which fits you best"	1

5.1 State-of-Practice and Expected Advancements

The discussion around the state-of-practice for the pattern selection was driven mostly by experienced participants that had some familiarity with patterns. Out of the 9 participants that claimed at least medium experience with patterns, 55% mentioned that when applying a pattern, they do it based on their experience, without having a look at additional resources (e.g., books, or online sources). One participant mentioned a mixed approach, i.e., shortlisting a couple of patterns, based on experience and then check their scope and structure in online resources. The rest 33% always checks online resources and attempts to get knowledge and familiarity from there, before selecting which pattern to apply.

Upon the experimentation with the tool, the practitioners have identified several advancements that the specific approach and accompanying tool can bring to their way of working. First, almost all developers mentioned that use of the approach can SAVE TIME from development. This can be achieved in various ways: (a) through ***code generation***, which can automate some of the straightforward tasks; (b) through the ***EXPERT-MODE*** the developer saves time for selecting the patterns, since all of them are presented together and the navigation among them is easy. Also, all the novice engineers mentioned that both ***EXPERT-MODE*** and ***WIZARD-MODE*** can act as a SOURCE OF KNOWLEDGE, since the former helps you to learn about patterns through the examples, the diagrams, and the brief scope; whereas the later can help you learn based on the key questions that you need to ask to yourself before applying a pattern. Furthermore, some participants mentioned that the tool can be useful to STAY ON TRACK, and not get lost in the many alternatives that exist, as well as within the vast number of resources that exist in the web. For achieving this benefit, a very important parameter is the fact that the flow of the tool is very close to the human way of thinking. Finally, one of the most experienced engineers in the company mentioned that the approach and tool can be very useful for EXPERIMENTATION purposes: "*Through the tool, the software engineer can practice some*

tentative design solutions, generate the code without any cost, and select which one fits the purpose of the design best. Design is a try-and-error process in any case".

The current state-of-practice in pattern selection usually relies on experience and online resources. However, not all software engineers have enough experience, and the amount of available resources might be confusing. Given these limitations, the SmartCLIDE pattern selection approach can advance the state-of-practice since it can aid novice developers in their decisions, train them, act as a learning material, and educate them through a trial-and-error experimentation in proper decision making.

5.2 Correctness, Timeliness, and Usefulness of the SmartCLIDE Pattern Selection Approach

By quantitatively comparing the correctness of the two modes of operation for SmartCLIDE Pattern Selection Approach, we can observe that the correct pattern was selected in 60% of the cases for both the ***EXPERT-MODE*** and the ***WIZARD-MODE***. However, the completion time for the ***WIZARD-MODE*** was substantially lower (approximately 8.5 minutes) compared to the ***EXPERT-MODE*** (17.8 minutes)—this difference has been characterized as statistically significant based on the results of a paired samples Wilcoxon test. This finding has validated the feeling of the practitioners (see Section 5.1) on SAVING TIME. Additionally, by focusing on the kind of errors in pattern selection identified in each mode we can observe that for the ***EXPERT-MODE*** only two mistakes were alternatives and the generated code could have led to a proper delivery of functionality (`State` instead of `Memento` and `Factory Method` instead of `Builder`), whereas for the ***WIZARD-MODE*** all errors have led to code that could be functionally correct (`Abstract Factory` instead of `Factory Method`, `Builder` instead of `Abstract Factory`, `Strategy` instead of `Bridge`, `State` instead of `Bridge`, `Composite` instead of `Decorator`). For instance, when using the `Strategy` pattern, instead of `Bridge`, the 2nd problem parameter instead of being placed in a 2nd hierarchy (`Bridge`), can be placed as an attribute in the only `Strategy` hierarchy. In that case, the polymorphic implementation of the `strategy method` will include an `if-statement` for handling the 2nd problem parameter. Although this solution is suboptimal, it can still produce working code. Finally, from the task analysis we have observed that the participants were marginally more confident when using ***WIZARD-MODE*** (~4.2 on average) compared to when using the ***EXPERT-MODE*** (~4.0 on average). However, this difference was not statistically significant.

In that sense, we can argue that the ***WIZARD-MODE*** helped more the developers to STAY ON TRACK, whereas the freedom that the ***EXPERT-MODE*** provided, worked better only for the experienced software engineers.

Additionally, to quantitatively assess the perceived usefulness of the main features of the SmartCLIDE pattern selection approach, we have applied a point system on the answers of the post-study questionnaire responses. To aggregate the scores from the 15 participants, we added the value that they assigned (1: not useful at all – 5: very useful). To improve the readability of the results in Figure 12, we have depicted the total points of each feature, as a percentage of the

75 total points that would have been awarded to the feature if all participants had graded it with 5 points.

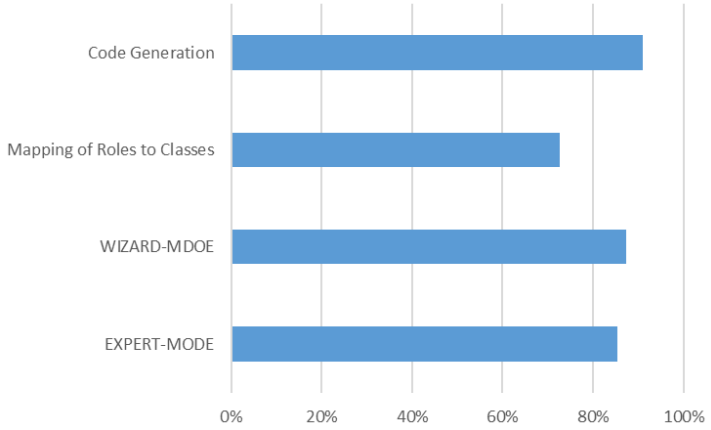


Figure 13. *Features Usefulness*

Next, we present the results of the qualitative analysis on the response of the participants in the focus group, related to RQ_2, so as to supplement and help the interpretation of the aforementioned findings. First, with respect to the ***EXPERT-MODE***, the developers have found the examples and the class diagrams as very useful, since the visualization has helped them to understand the pattern, even without extensive prior knowledge (SOURCE OF KNOWLEDGE). On the other hand, some participants mentioned that the tool (to achieve an industrially-ready solution) must hide the complexity of pattern TERMINOLOGY, since especially junior developers struggle to understand the notions of the pattern language.

With respect to the ***WIZARD-MODE***, the developers found the questions straightforward and were able to lead the participants to the pattern with confidence (DECISION CONFIDENCE). However, an interesting suggestion on the Q&A process was made from a novice software engineer: "*It would be great to take no previous knowledge for granted. For instance, I was not confident even for the type of the pattern that I need to use: Creational, Behavioral, or Structural*". Also, almost all participants mentioned that this operation mode was substantially faster (SAVE_TIME), whereas the novice software engineers noted that the Q&A can guide us more easily that internet. An interesting observation that came out of the focus group, highlighting that the approach helps developers to STAY ON TRACK was an example of a developer who picked a wrong pattern (`Decorator` instead of `Composite`), explained as below: "*I remember that I have seen a similar example in the internet, and I wanted to lead the tool to the* `Decorator` *pattern. But the Q&A process did not allow me to navigate there, it led me to* `Composite`*. I was not satisfied that the tool did not give me freedom to pick the pattern that I wanted*!"[8].

The **Code Generation** feature was the only one with no negative discussion around it. The main usefulness discussed for this feature was the SAVED TIME, and that this feature was an integral part of the solution, linking patterns to code, which is the final outcome of the designing

8 The task was inspired by the example that the participant mentioned, but it was altered by the researchers so as to better fit `Composite` rather than `Decorator`.

process. Therefore, not ISOLATING DESIGN FROM CODING process and environment. Such options (being in favor of integrating development aspects in the IDE) are very popular among developers, and can be identified in other similar studies (Charalampidou et al., 2018). Another interesting position was that the integrated code generation will help the developers avoid copying and pasting solutions from the internet that then will then need to be stripped out of useless code. The code that the code generation provides is the MINIMUM REQUIRED CODE on top of which you can develop the business logic around the pattern. Finally, the code generation can be perceived as a feature that will enable CODE READABILITY, by guiding in terms of styling, best practices. This can contribute to more readable code, on top of the more maintainable design.

Finally, in terms of **Mapping Pattern Roles to Classes** a lot of useful feedback has been received, since almost all participants found it difficult to map roles to classes. However, the mapping step cannot be removed, since it is a pre-requisite for **Code Generation**. An interesting suggestion from a senior engineer was to "*make the UI more interactive in that part, and enable the selection of the role from the example class diagram, so that the visual information is exploited*". Additionally, the participants raised a well-known problem in object-oriented programming, dealing with the difficulty in identifying proper names of the classes, especially in such an early stage (Allamanis, 2015). Also, some participants were puzzled to identify which roles correspond to classes, methods, or attributes, bringing up the TERMINOLOGY problem. On the positive side, the participants recognized that such a mapping can preserve the application of pattern rules contributing towards CORRECTNESS of the implementation, and since the process has a LOW LEARNING CURVE it can also educate developers on TERMINOGY issues (SOURCE OF KNOWLEDGE).

The participants ranked **Code Generation** as the most useful part of the solution, a fact that underlines their satisfaction from SAVING TIME. The ***WIZARD-MODE*** was slightly more popular, compared to the ***EXPERT-MODE***, a result that can be attributed to our dataset that involved more junior, compared to experienced software engineers. Finally, the participants have found the use of **Mapping of Roles to Classes** as very complicated.

5.3 Usability Evaluation

The usability of the SmartCLIDE pattern selection Theia extension has been positively evaluated, with an average grade B (73.3%), ranging from D (min: 55) to A (max: 90)—see Figure 8. The frequency of D grades was 13%, whereas 40% of the participants evaluated the Theia extension as A-class. By studying isolated questions, the extension seemed very CONSISTENT to the users and of LOW COMPLEXITY. One participant vividly described that: "*the tool is very easy to use, the theme is consistent to the general layout of Theia, I could use it without any guidance*". On the other hand, the most negative evaluations have been received with respect to the LEARNING CURVE and the NEED FOR SUPPORT / MANUAL. In particular, some practitioners mentioned that "*someone needs to first read on patterns, and then use the tool. In that sense, I have a lot of reading to do, before using it efficiently*", whereas another mentioned that "*a help button is a must have for modern applications*".

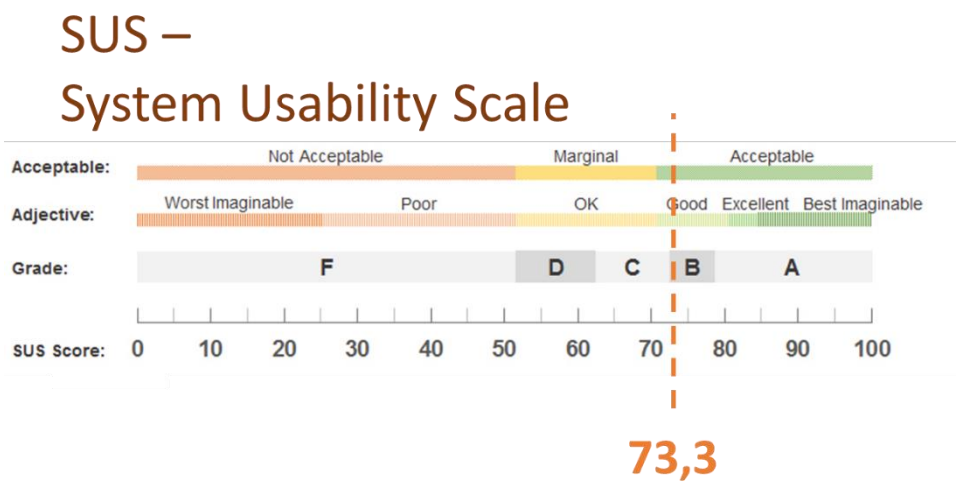


Figure 14. *Usability Evaluation Outcome*

The developed Eclipse Theia extension for aiding in pattern selection has received a positive evaluation in terms of usability, constituting it acceptable for industrial usage.

6. Discussion

The proposed approach and tool support only the GoF patterns, whereas other pattern types exist in the literature. The decision to focus on the GoF patterns was taken as these patterns are the most popular ones and the primary means to understand the concept of patterns in software design. If a development team wishes to adopt the proposed approach and extend the use to a wider set of patterns, the corresponding decision trees can be enriched, assuming the required domain knowledge. Furthermore, while the proposed tooling is capable of instantiating the patterns selected by the guided interaction with the user, the tool does not rely on the context of the target system. In other words, the approach lacks any sophisticated intelligence to infer the patterns that might be more relevant to the user's needs. While the introduction of AI to limit the number of questions that have to be answered by the end user is beyond the scope of this work, we believe that appropriate Machine Learning algorithms could be leveraged to recommend potential pattern solutions based on similar code retrieved from repositories.

Regarding the industrial validation of the proposed approach which has been performed in the context of a single company with the help of 15 engineers, the results unavoidably reflect the environment and practices of the particular company and the experience and expertise of the selected participants. Consequently, the findings are subject to generalizability threats; however, since the goal was not to compare the proposed approach against similar techniques but rather to investigate its potential and weaknesses, we believe that the quantitative and qualitative analysis shed light into the effectiveness of a pattern selection approach that is based on structured questions. Nevertheless, further studies on the usability of the corresponding Eclipse Theia plugin could reveal optimization in the interaction with end users.

Considering that part of the evaluation consists in a qualitative study, respondent bias should be taken into account. Respondent bias refers to cases where participants do not provide honest responses usually stemming from the willingness to ‘please’ the researcher with responses they believe are desirable (Lincoln & Guba, 1985). Qualitative studies of this kind are also threatened by reactivity, referring to the possible influence of the researcher on the studied participants. An enthusiastic researcher might have affected the participants of a focus group by steering the discussions to a particular stance. While this sort of bias cannot be eliminated, method triangulation has been applied to increase the validity of the findings on all three RQs and thereby reduce the corresponding threats (Robson, 2002).

7. Conclusions

Design Patterns, as general, documented and repeatable solutions to commonly occurring problems in software design can promote good software development and increase maintainability and extensibility. However, the application of patterns is not trivial: the choice of the most suitable pattern is not always obvious whereas often a no-pattern solution is preferable. The correct instantiation of patterns also poses challenges, especially for design alternatives with marginal differences. To ease the work of software engineers and encourage the consideration of design patterns in everyday software development, we introduce a questionnaire-based approach relying on decision trees that guides end users in the selection of the proper design pattern. The functionality is provided through an Eclipse Theia plugin that is capable of generating and integrating the pattern code with the rest of the codebase.

An industrial validation study employing questionnaires, focus groups, and task analysis was carried out with the help of 15 software engineers. The results suggest that a structured interaction with the end user increases the probability of selecting the proper design pattern and saves development time. Furthermore, a tool that interacts with the users providing examples can act as a source of knowledge and educate developers on the rather challenging topic of design pattern application. Future research can investigate ways to increase the usability of the design pattern selection tool and the possibility of leveraging AI techniques for limiting the number of questions that have to be set to the user for deciding on the most appropriate design alternative.

Reference

Ampatzoglou A. (2012), “Επίδραση των Προτύπων Σχεδίασης στην Ποιότητα Λογισμικού”, PhD Thesis, Aristotle University Of Thessaloniki, Thessaloniki.

Ampatzoglou A. , Chatzigeorgiou A., Charalampidou S. and Avgeriou P., (2015). “The Effect of GoF Design Patterns on Stability: A Case Study”, Transactions on Software Engineering,

Ampatzoglou, A., Charalampidou, S. and Stamelos, I.(2013), “Research state of the art on GoF design patterns: A mapping study”, Journal of Systems and Software, 86 (7) pp. 1945-1964.

Andhariya A. (2017). Observer Design Pattern [Online]. Available at: https://codepumpkin.com/observer-design-pattern/ (Accessed: 10 June 2022).

Arvanitou E.M. (2011) ‘Εμπειρική Μελέτη της Επίδρασης των Προτύπων Σχεδίασης στα σφάλματα Λογισμικού Παιχνιδιών’, Master Thesis, T.E.I Of Thessaloniki

Arvanitou, E.M. (2011). Class Diagram of Factory Method, Thessaloniki.

Arvanitou, E.M. (2011). Class Diagram of Observer, Thessaloniki.

Arvanitou, E.M. (2011). Class Diagram of Strategy, Thessaloniki.

Baeldung (2022). The Decorator in Java [Online]. Available at: https://www.baeldung.com/java-decorator-pattern (Accessed: 9 June 2022).

Bishop, J. (2008), Language features meet design patterns: raising the abstraction bar”, 2nd International Workshop on the role of abstraction in software engineering (ICSE’08), IEEE, pp. 1-7, Leipzig, Germany.

Briand, L. C., Labiche, Y. and Sauve, A. (2006), “Guiding the Application of Design Patterns Based on UML Models”, 22nd International Conference on Software Maintenance, IEEE, pp. 234-243, Philadelphia, Pennsylvania.

Brooke, J. (1996). “System Usability Scale (SUS): A quick-and-dirty method of system evaluation user information”, Taylor & Francis, pp. 189-194.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). “Pattern-Oriented Software Architecture”, Wiley & Sons, West Sussex, UK.

Charalampidou, S., Ampatzoglou, A., Chatzigeorgiou, A. and Tsiridis, N. (2018), “Integrating traceability within the IDE to prevent requirements documentation debt”, 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA ’18), IEEE, pp. 421-428.

Cinneide, M. O' and Nixon, P. (2001), "Automated software evolution towards design patterns", 4th International Workshop on Principles of Software Evolution (ICSE'01), IEEE, pp.162-165, Vienna, Austria.

EclipseSource (2019). Eclipse Theia extensions vs. plugins vs. Che-Theia plugins [Online]. Available at: https://eclipsesource.com/blogs/2019/10/10/eclipse-theia-extensions-vs-plugins-vs-che-theia-plugins/amp (Accessed: 28 June 2022).

Elo, S. and Kyngäs, H. (2008), "The qualitative content analysis process", Journal of Advanced Nursing, vol. 62, issue 1, pp. 107-115.

Feitosa, D., Avgeriou P., Ampatzoglou A. and Nakagawa E. Y. (2017). "The evolution of design pattern grime: An industrial case study", International Conference on Product-Focused Software Process Improvement (PROFES '17), Springer, pp. 165-181.

Finch D. (2020). Decorator Design Pattern Explained – Structural Design Patterns [Online]. Available at: https://darrenfinch.com/decorator-design-pattern-explained-structural-design-patterns/ (Accessed: 11 June 2022).

Fowler, M. (1996, "Analysis patterns: Reusable object models", Addison-Wesley Professional.

Gamma, E., Helms R., Johnson, R. and Vlissides J. (1995). "Design patterns: elements of reusable Object-Oriented software", Addison-Wesley Professional.

Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. M. (1994), Design Patterns: Elements of Reusable Object-Oriented Software , Addison-Wesley Professional .

Gupta, L. (2021). Bridge pattern participants, HowToDoInJava, https://howtodoinjava.com/design-patterns/structural/bridge-design-pattern/.

Hsueh, N.L., Chu P.H. , Hsiung, P.A, Chuang, M.J., Chu, W., Chang, C.H, Koong, C.S. and Shih, C.H. (2010), "Supporting Design Enhancement by Pattern-Based Transformation", 34th Annual Computer Software and Applications Conference (COMPSAC '10), IEEE, pp. 462 – 467, Seoul, Korea.

Keepence, B. and. Mannion, M.(1999), "Using Patterns to Model Variability in Product Families", IEEE Software, IEEE, 16 (4), pp. 102-108.

Krasnoshchok, D. (2022). autocompleter [Online]. Available at: https://www.npmjs.com/package/autocompleter (Accessed: 5 April 2022).

Lincoln, Y. and Guba, E. G. (1985), "Naturalistic Inquiry", Newbury Park, CA: SAGE.

M. Allamanis, E. T. Barr, C. Bird, and C. Sutton (2015), “Suggesting accurate method and class names”, 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE ‘15), ACM, pp. 38–49.

MacDonald, S., Szafron, D., Schaeffer, J., Anvik, J., Bromling, S. and Tan, K. (2002), “Generative Design Patterns”, 17th IEEE International Conference on Automated Software Engineering (ASE ‘02), pp. 23, Edinburgh, UK.

MacDonald, V, Tan, K., Schaeffer, J. and Szafron, D. (2009), “Deferring Design Pattern Decisions and Automating Structural Pattern Changes Using a Design-Pattern-Based Programming System”, Transactions on Programming Languages and Systems, ACM, 31(3), article 9..

Martin, R. C. (2003), “Agile software development: principles, patterns, and practices”, Prentice Hall PTR, Upper Saddle River, USA.

Mayvan, B. B., Rasoolzadegan, A and Yazdi, Z. G. (2017), “The state of the art on design patterns: A systematic mapping of the literature”, Journal of Systems and Software, 125, pp. 93-118,

Meyer, M. (2006), “Pattern-based Reengineering of Software Systems”, 13th Working Conference on Reverse Engineering, pp.305-306, Benevento, Italy.

Patra R. (2020) Factory Method Design Pattern In C# [Online]. Available at : https://www.c-sharpcorner.com/article/factory-method-design-pattern-in-c-sharp/ (Accessed: 9 June 2022).

Refactoring Guru (n.d). Strategy [Online]. Available at: https://refactoring.guru/design-patterns/strategy (Accessed: 10 June 2022).

Refactoring Guru (n.d). Strategy in Java [Online]. Available at:https://refactoring.guru/design-patterns/strategy/java/example (Accessed: 10 June 2022).

Robson, C. (2002), “Real world research: a resource for social scientists and practitioner-researchers”. Oxford, UK: Blackwell Publisher.

Runeson, P., Host, M., Rainer, A. and Regnell, B.(2012), “Case study research in software engineering: Guidelines and examples”, Wiley & Sons, West Sussex, UK.

Seaman, C. (1999), “Qualitative Methods in Empirical Studies of Software Engineering”, IEEE Transactions on Software Engineering, 25 (4), pp. 557–572.

Shalloway, A. and Trott, J.(2004), “Design Patterns Explained: A New Perspective on Object Oriented Design”, Addison-Wesley, 2nd Edition (Software Patterns).

SourceMaking (n.d). Builder Design Pattern [Online]. Available at: https://sourcemaking.com/design_patterns/builder (Accessed: 9 June 2022).

SourceMaking (n.d). Builder In Java [Online]. Available at: https://sourcemaking.com/design_patterns/builder/java/2 (Accessed: 9 June 2022).

SourceMaking (n.d). Command Design Pattern [Online]. Available at: https://sourcemaking.com/design_patterns/command (Accessed: 9 June 2022).

Spencer, D. (2009), "Card Sorting: Designing Usable Categories". Rosenfeld Media, 1st Edition.

Theia (n.d). Extensions and Plugins [Online], Available at: https://theia-ide.org/docs/extensions (Accessed: 28 June 2022).

Theia (n.d). Widgets [Online], Available at: https://theia-ide.org/docs/widgets/ (Accessed: 28 June 2022).

Tonella, P. and Antoniol, G.(2001), "Object Oriented Design Pattern Inference", Journal of Software Maintenance and Evolution, Wiley, 13 (5).

Trashtoy (2006). UML class diagram for Composite software design pattern. Wikimedia Commons, https://commons.wikimedia.org/wiki/File:Composite_UML_class_diagram.svg

Trashtoy (2007). UML class diagram for Bridge software design pattern. Wikimedia Commons, https://commons.wikimedia.org/wiki/File:Bridge_UML_class_diagram.svg

Trashtoy (2007). UML class diagram for Builder software design pattern. Wikimedia Commons, https://commons.wikimedia.org/wiki/File:Builder_UML_class_diagram.svg

Wikipedia (n.d), UML diagram of the command pattern, https://en.wikipedia.org/wiki/Command_pattern

Yau, S. S. and Dong, N. (2000), "Integration in Component-Based Software Development Using Design Patterns", 24th International Computer Software and Applications Conference (COMPSAC'00), IEEE, pp.369, Taipei, Taiwan, 25-28 October