SmartCLIDE Design Pattern Assistant: A Decision-Tree based Approach

Eleni Polyzoidou[1], Evangelia Papagiannaki[1], Nikolaos Nikolaidis[1], Apostolos Ampatzoglou[1], Nikolaos Mittas[3], Elvira Maria Arvanitou[1], Alexander Chatzigeorgiou[1], George Manolis[2], Evdoxia Manganopoulou[2]

[1] *Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece*
[2] *Onelity, Greece, SMPC*
[3] *Department of Chemistry, International Hellenic University, Kavala, Greece*

nnikolaidis@uom.edu.gr, a.ampatzoglou@uom.edu.gr, e.arvanitou@uom.edu.gr, achat@uom.edu.gr, nmittas@chem.ihu.gr

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 (Gang-of-Four) 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 [1], Architectural Patterns [2], and Design Patterns [3]. 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 point of views (e.g., effect on quality, applicability, automated detection, etc. [4][5]), 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 use polymorphism to capture 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 [6].

[p3] ***improper instantiation.*** In some cases, the coding of a GoF design pattern solution is not 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 [7][1]. In other words, the proposed approach provides a sequence of questions that when answered by the software engineer, can guide them either to a fitting pattern for the specific design problem, or to advise them not to use a pattern. Given the fact that GoF patterns are applied to functional requirements (they are not horizontal to the whole system, e.g., as architectural patterns are), we anticipate that the software engineer will apply the method (and use the tool) in various cases of the software design process. More specifically, we anticipate a software engineer to use the tool, when: (a) he/she has a hint that a pattern might be applicable; or (b) he/she believe that the specific requirement to be designed will be part of the design hotspot, and urges for an extensible design solution. Finally, we need to note that the code generation is restricted to the initial pattern instantiation: i.e., class declarations, attributes, and methods that are required to instantiate the pattern, and not the actual implementation of the functional requirement, since that goes beyond the design decision-making phase.

The tool has been developed as part of the SmartCLIDE project[2], is released as an Eclipse Theia[3] extension that is stored and distributed through Eclipse Research Labs `git` repositories[4]. In contrast to static documentation (such as books and an online sources), we believe that the proposed interactive Q&A approach will be more helpful, in the sense that: (a) it will provide a just-in-time source of knowledge for the developer, who can consult it when needed; (b) it will be a starting point to make decisions for developers that are not aware of patterns, or that are not experts in patterns and do not recall all the required information; (c) it will save time from searching in documentation and re-write or copy paste material and initial source code solutions from the web that require a lot of tailoring to fit the context of the specific requirement at hand—leading to less errors, sloppy code, and pattern grime. To evaluate the proposed approach and the accompanying Eclipse Theia plugin, we have conducted an industrial validation with an SME that is independent to 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.

---

[1] We note that the code generated by the proposed approach follows the original guideline of the GoF book. In that sense it does not cover any pattern alternatives, or variants [50]. This decision was made to ensure the proper instantiation of the pattern; however, it does not prohibit the developer to deviate from this.

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

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

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

2. Related Work – Supporting the Application of Design Patterns

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. [5], "***GoF design pattern application"*** is the core research sub-topic on patterns. The topic involves research endeavors presenting methods for identifying systems that need pattern application or methods and tools that automate or assist the selection and the application of patterns. The research landscape in this direction can be organized into 6 main categories: (a) design pattern selection; (b) design pattern abstraction; (c) re-engineering to patterns; (d) generative design patterns; (e) automated code transformation; and (f) pattern-based architecture. Each one of the aforementioned lines of research are described in detail below.

2.1 Design Patterns Selection

The ***Design Pattern Selection*** constitutes one of the greater barriers to the application of the design patterns and the core of the current research. Palma et al. [45] introduce the development of an expert system - Design Pattern Recommender (DPR) – based on four steps: (a) knowledge acquisition and verification from literature on object-oriented software design patterns, (b) definition of the conditions of each design pattern and refinement of them into sub-conditions, (c) turning all the sub-conditions into nodes of a tree for each pattern and convert each node to question, (d) building a Goal-Question-Metric (GQM) model – measurement of the designers' answers weight and calculation of every answer's points. Finally, the points for every pattern are summarized and the pattern with the most points is suggested. The DPR appears promising, but results have shown that is more efficient with a few additional features. A more strategic perspective about the design pattern selection solution has given by Sahly and Sallabi [49]. The aforementioned authors, as first step, classify the user's levels by Dreyfus model – Novice, Advanced beginner and Expert. Next step is the identification of the design problem by the user in order for him/her to obtain the suitable patterns. Later, the retrieval of suitable patterns takes place. The strategies for this step are four – QMP algorithm, QSPQ algorithm, CIK algorithm and QAS algorithm - and executed subsequently if necessary. The scenario of execution is the following: the user submits a query and QMP algorithm runs. If no appropriate patterns were founded, then QSPQ algorithm is executed. If, again, no appropriate patterns were founded, then if the user is Expert, CIK algorithm is executed, whereas if the user is Novice or Advanced beginner, then the QAS algorithm is executed. Finally, if QAS algorithm haven't found any relevant results, CIK algorithm is executed. Query-Matching-Pattern (QMP) parses and analyzes the text user's query in order to find matching between patterns intents and the given query. Query-Similarity-Previous Query (QSPQ) searches the similarity between the requested query and previous users' queries to find recommended patterns. Question-Answer-Session (QAS) is based on a question – answer session. The questions vary accordingly to the previous answers of the user, weights are calculated for the recommendations and the proposed patterns are narrowed down. Collaborative-Implicit-knowledge (CIK) retrieves knowledge from communities of users. When CIK receives a request from the user with the description of the design problem, the request is intercepted by the observer and the observer sends the query to the database for users interested in the same domain. Subsequently, all suggestions/solutions of the problem presented are sent backwards to the observer and the user. This approach can assist developers on design pattern selection, by offering them a guide and the wider knowledge of a community.

Naghur and Hasheminejaddipo [48] use ontology in order to create a knowledge model for design pattern use cases. The purpose of the ontology model is to support sharing and enable queries to datasets of abstract information. The tools that have been used are: (a) a semantic markup language ("OWL") - for publishing and sharing the ontologies, (b) a graphical user interface ("Protégé"), (c) a free, open-source Java framework for building semantic web and linked data ("Jena") – for managing the ontologies and (d) a query language ("SPARQL"). The use cases are textual and consists of different sections such as name, actors, precondition, description of the main scenario, post-conditions, functional requirements, etc. and the the design pattern which solves the use case. Each use case is processed by Natural Language Processing (NLP). The actions that take place are tokenization, removal of stop words, word stemming, labeling the words based on their role and, finally, dividing the use case into its components. When the use cases are in the proper format, the constructed ontology is questioned

using design problems to select the appropriate design pattern. The correctness of the results is 96.55%, much higher than other approaches. Rahmati et al. [46] proposed an approach that is based on a vector space model (VSM) and the explicit semantic analysis (ESA); both belong to the best methods of information retrieval. The ESA is used for determining the similarity between the user input text and definition texts of patterns, after is indexed on all the Wikipedia papers. There are two ways for finding the similarity: firstly, by comparing the text input to text from the GoF book and, secondly, by comparing the text input to a set of keywords from the GoF book by using the "Alchemy API". For the evaluation of the text, a dictionary ("WordNet") is used, in order to extract synonyms of the words in the text. After WordNet has created groups of synonyms, the Term Frequency-Inverse Document Frequency (TF-IDF) weighting algorithm, which has been improved by the researchers, determines the degree of similarity between the input text and the definition text of each design pattern: vectors of the texts are created according to the weights and the angle is calculated between two vectors, based on the VSM. The results of the approach show high numbers on precision and recall.

Pavlič et al. [44] use ontology for obtaining knowledge on the design patterns and a set of question/answers pairs that indicate the applicability of design patterns. They, also, present a usability function, which applies the weights of the expert's answers and an algorithm, that is used for the selection of design patterns. If the adequate description of the design problem is considered granted, then the level of usability is calculated and the appropriateness of each specific design pattern is determined. The set of the answered questions is also used to determine the next most relevant question, until the usability of the most appropriate design pattern is dominant over all the other patterns. Through this approach, information relevant to design patterns can be shared and automatically analyzed. In combination with the Design Pattern Expert (DPEX) tool, the developers were capable of solving more problems than without the tool. Finally, Naghdipour et al. [48] suggest a more generic approach, a framework called "DPSA", which includes the classification of other relative approaches on pattern selection and the analysis of these, according to certain criteria. The presentation of the patterns starts from the definition of the targeted group–the most usual is the GoF design patterns. The presentation of the patterns is followed by the definition of the design problem. The analyzing method contains the comparison between the suggested pattern(s) by the approach and the design problem. The suggested pattern is the one with the most similarities.

2.2 Design Patterns Abstraction

The ***Design Patterns Abstraction*** research topic includes studies suggesting that design patterns are the key to provide abstraction in software and for adapting software components into existing systems. Bishop [8] 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, they 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 [9] 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. Additionally, Yau and Dong [10] 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 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: to develop a chatting room, which is used for several people in one group to talk simultaneously.

2.3 Re-Engineering to Patterns

The ***Re-engineering to Patterns*** research topic includes studies that propose methods for detecting software anti-patterns that necessitate re-engineering through design pattern application. Briand et al. [11] 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 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 can 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. Furthermore, Meyer [12] provide 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.

2.4 Generative Design Patterns

Generative Design Patterns correspond to techniques that aim to automatically generate design pattern instances. MacDonald et al. [13] 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 research group [14] 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 affecting 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.

2.5 Automated Code Transformation

The ***Automated Code Transformation*** research topic includes studies that propose methodologies for automatically constructing transformations that can be used to apply GoF design patterns. O' Cinneide and Nixon [15] 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 behavior 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 behavior-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 can apply these pattern transformations to a Java program. Finally, they used an example of the application of the Factory Method transformation to a generic program. The authors applied the methodology to a set of patterns from the GoF catalogue, 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. Moreover, Hsueh et al. [16] 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: (a) pattern description, (b) functional requirement intent, (c) non-functional requirement intent, (d)

functional requirement structure, (e) non-functional requirement structure, and (f) transformation specification. For the automating pattern application, Hsueh et al. [16] 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.

2.6 Pattern-based Architectures

Tonella and Antoniol [17] 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 containing 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 pattern instances is an indicator of the reuse frequency of the identified class organization, while the number of involved relations represents the complexity of the pattern. To evaluate their approach, Tonella and Antoniol [17] performed a case study on C++ applications. They first examined the methods that 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.

3. Background Information

3.1 Design Decisions in Architecture

Software architecture is defined at the beginning of a project and it largely affects the success of development and maintenance [28]. For this reason, a number of studies have been carried out to support architects in their decision-making activities. Falessi et al. [29] proposed a systematic way to choose among decision-making techniques for resolving trade-offs in architecture design. By comparing the top existing techniques for decision-making, they found out that there is no perfect decision-making technique, but one should choose the best one that suits his/her needs in each case. To this end, they created a characterization schema that can select one of 15 existing decision-making techniques, with the use of some quality attributes as selection criteria. Similarly, Shahin et al. [30] listed a lot of existing methodologies and tools and pointed out their differences in order to help the architects of a given system. Moreover, Kazman et al. [31] introduced the Architecture Tradeoff Analysis Method (ATAM) to provide a methodology to help architects rationally select the best architecture for their system in a given state. The ATAM is a robust methodology that provides the tradeoffs of each different option and can also help better clarify the requirements of the software in an earlier state. Finally, Van Vliet and Tang [32] pointed out the main issues of software architecture design and how important the earlier correct decision for a system is, as well as the usefulness of reflective questions during the process of the decision. Another popular approach in the architecture domain is the scenario-based strategy. Ionita et al. [33] used the Strategic Scenario-Based Architecting in order to make decisions about the future architecture of a system. With this approach, there is continuous feedback regarding the future state of the system as the four main steps that were introduced run continuously in a circle. This methodology was also used in a case study for the selection of better decisions throughout the course of a project, with very good results. Golfarelli et al. [34] examined many different tools that are being used for the decision-making process, which have as common denominator the what-if analysis. The study focuses on the lessons learned by the usage of the what-if analysis and provides a new methodological framework based on their results.

3.2 Case Study Design Methodologies

The first guidelines for case study design were published by Kitchenham et al. [35], focusing on the use of quantitative data; being followed by Seaman [36] who published guidelines for qualitative research. In 2000, a

broader set of guidelines on empirical research was published by Kitchenham et al. [37], who suggested guidelines for basic research topics: (a) experimental context, (b) experimental design, (c) conduct of the experiment and Data collection, (d) analysis, (e) presentation of results, and (f) interpretation of results. In a more industry-driven context, Verner et al. [38] focused on the early phases of a detailed plan design; organizing the case study guidelines into seven phases: (a) research initiation or pre-planning, (b) administration, (c) focus case study or planning, (d) design case study plan, (e) data collection, (f) data analysis, and (g) reporting. Additionally, they identified steps within each phase, however the ordering of the steps in each phase do not need always be sequential. Finally, Wohlin et al. [39] and Runeson et al., [19] recommended for case studies a five-step process: (a) case study design—objectives are defined and the case study is planned, (b) preparation for data collection—procedures and protocols for data collection are defined, (c) collection of data—execution of data collection on the studied case, (d) analysis of collected data— data analysis procedures are applied to the data, and (e) reporting— the study and its conclusions are packaged in feasible formats for reporting.

3.3 Usability Assessment Strategies

Usability is an important software quality, being related to the interaction of a system and a user [42]. The evaluation methods can be divided into the following categories [42]:

(a) **Inspection Methods** refer to the case where a group of specialists determine if a given user interface design follows specific guidelines that are being given to them (e.g., through expert review) [40]. The scope of the review is defined from the start and a list of issues that are related to the usability of the system, is the result of it. Its major strength is its low cost for resources; however, it is less reliable than usability testing [41];

(b) **Walkthroughs** are processes where a specialist is working with the development team, with the aim to find usability issues. Specialist's main responsibilities are the preparation of the user profiles, the tasks that are going to get analyzed by the developers, the questions that he is going to ask for each task and the rules for each walkthrough. Even though walkthroughs are not considered the most popular evaluation method (due to preparation need), they are useful since they do not need additional resources (only development team);

(c) **Usability Testing** is a process where the end-users have to complete a couple of predefined tasks using a system and the usability specialist records the results with the intent to identify usability issues [40]. In the assessment of a system using this method, participants are end users, unlike the participants of the inspection methods [40]. Users are requested to verbalize their thoughts, while performing a task or after the completion of it, because thinking aloud is a technique that has been used in psychological studies for a long time, and it helps gather cognitive information [43]. It is believed to be the assessment method which has the most impact on people. The drawbacks of this method are the time and resources that are required, and they are only identified easier problems [41]; and

(d) **Inquiry Methods** are methods that are frequently used, after usability testing, aiming to gather information about users' impression of a user interface [42]. Inquiry methods use several data collection methods, such as surveys, interviews, and focus groups. Surveys is a popular method to collect information with low cost [41]; interviews are applicable for situations, where there is a need for detailed answers from a small group of participants; and focus group the open discussion on usability issues, accompanied by task assignments [41].

3. SmartCLIDE Design Pattern Assistant

3.1 Proposed Approach

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 pinpoint to the most fitting ones. The methodology to construct the binary decision trees involved various iterations among pattern experts from both academia and industry. The steps of the methodology are presented below:

1. study the definitions and examples of GoF patterns, e.g., from books [3][18] and online sources[5,6]
2. compile sets of patterns that are alternatives, and a primary reason that leads to the selection of a pattern
3. review the aforementioned outcomes, by pattern experts from academia and industry partners
4. group the reasons to use a pattern, with most common reasons being closer to the root of the decision tree
5. transform the reasons to a Q&A format
6. 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, an additional expert was added. The review rounds were terminated when the additional expert had no supplementary feedback.

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.

Alternatives:

`Abstract Factory`, `Builder`, and `Factory Method`, in the sense that they all are able to handle the creation of objects from families of products

`Singleton` and `Prototype`, these patterns are used when you do not want to create multiple NEW objects, but reuse one or clone an existing to the new ones

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 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

`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 object (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 for gathering code generation information; 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.

[5] https://refactoring.guru/design-patterns
[6] https://sourcemaking.com/design_patterns

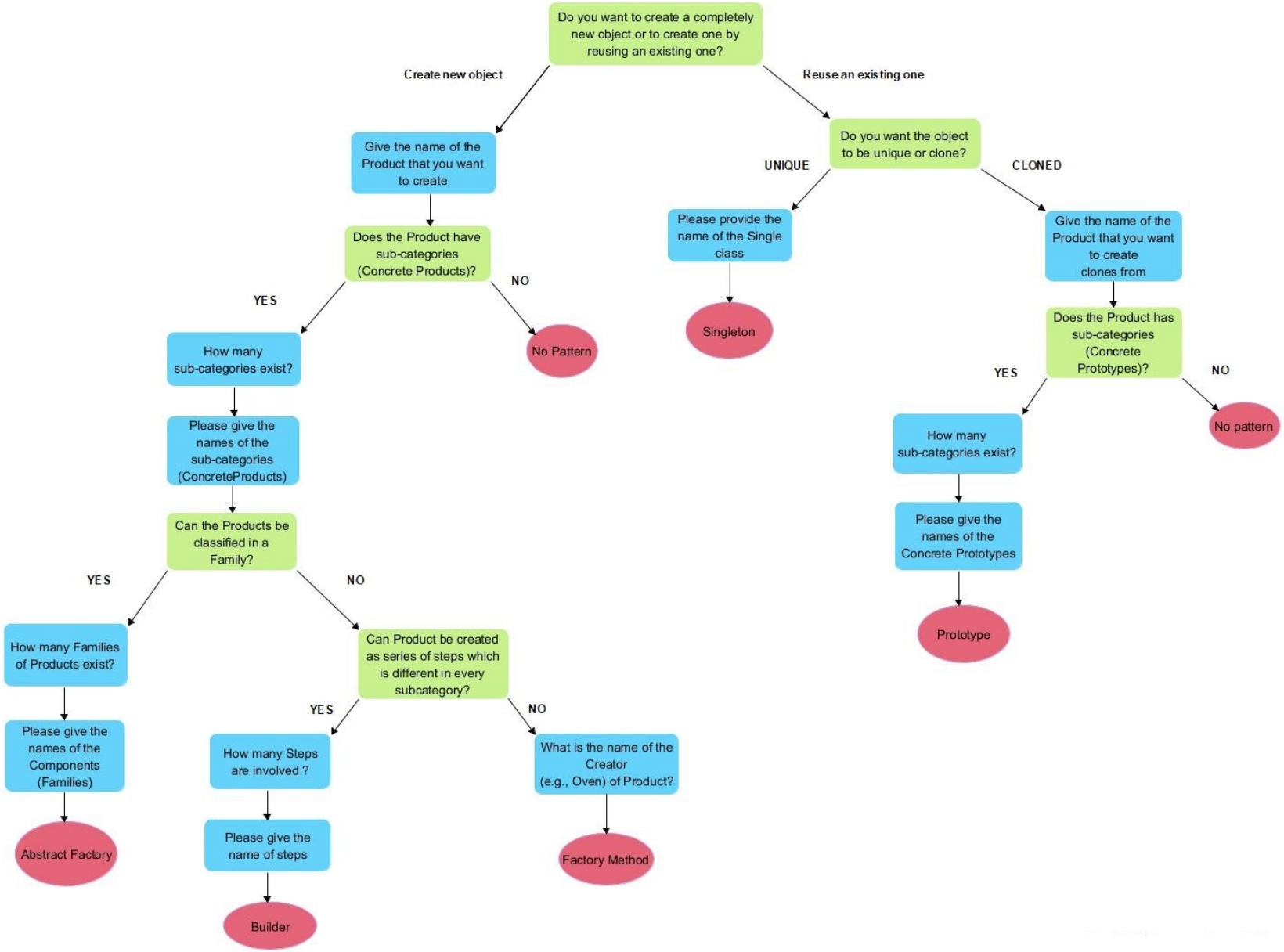

Figure 1. Decision Tree for the Selection of Creational Design Patterns

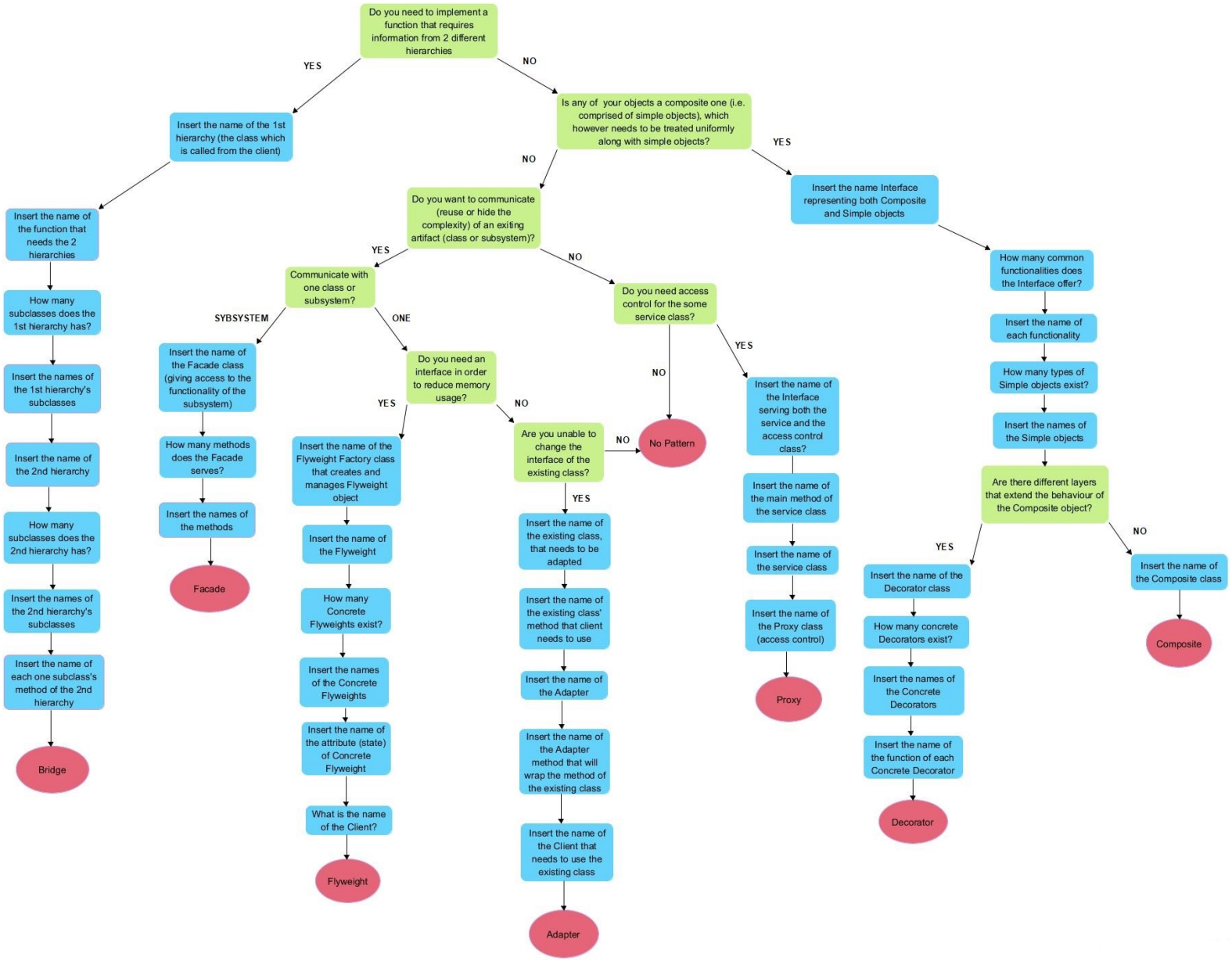

Figure 2. Decision Tree for the Selection of Structural Design Patterns

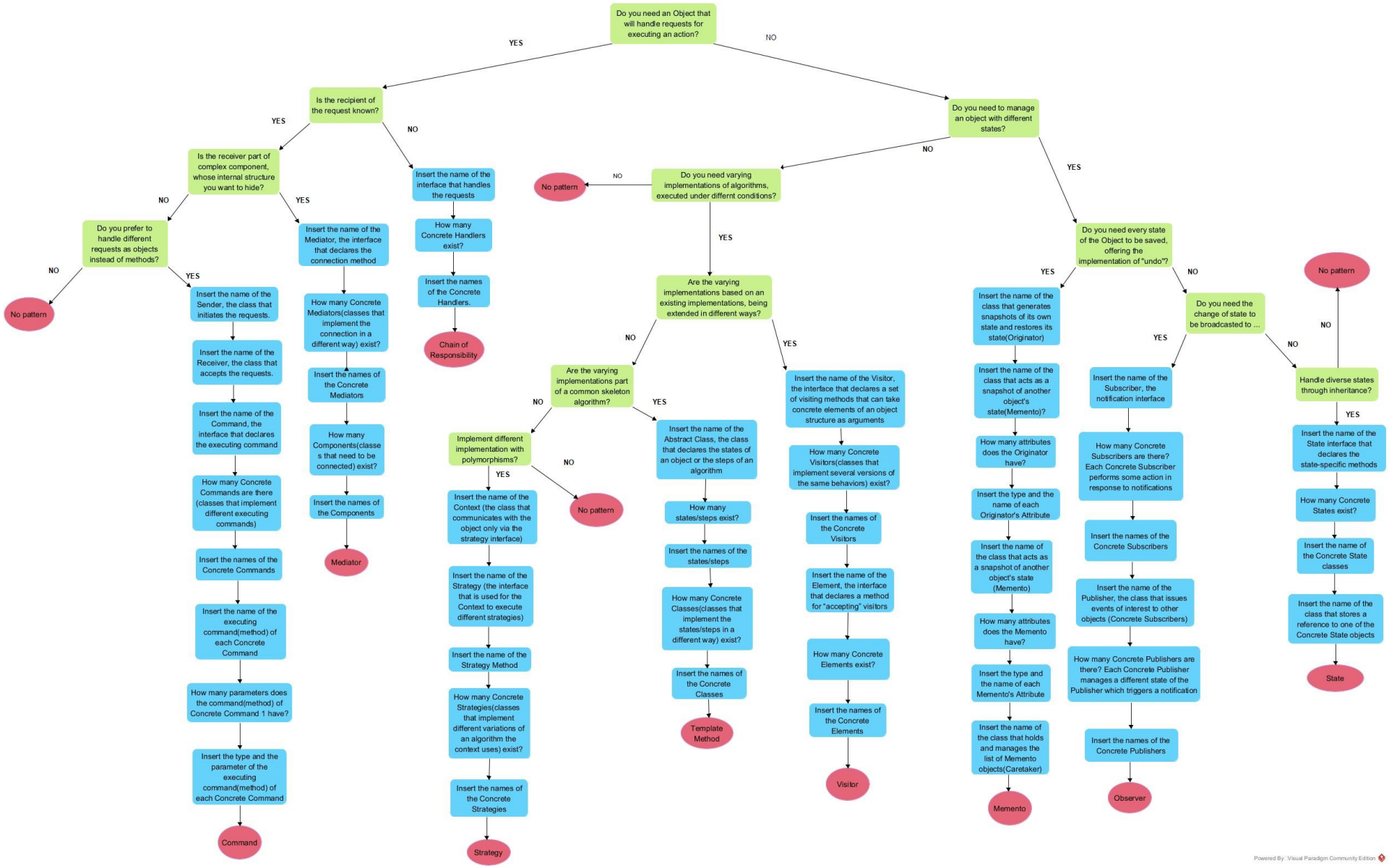

Figure 3. Decision Tree for the Selection of Behavioral Design Patterns

As an initial testing for the proposed approach, and before proceeding to the industrial validation, we have performed a proof-of-concept analysis, based on well-known open-source projects that have documented their pattern usage. In particular, similar to our previous work [52], we have explored the instances of design patterns in JHotDraw 5.1 and explored if the approach would have ended-up in the same results. The outcome of this analysis is presented in Appendix A.

3.2 Eclipse Theia Extension

To add the created extension in the SmartCLIDE platform, we build a new instance of Eclipse Theia, which is deployed as a Docker container[7]. The user is able to open the Design Pattern Assistant extension from the View menu item, appearing on 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 SmartCLIDE platform[8]. In the right part of the figure, we can see 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 (**WIZARDMODE**).

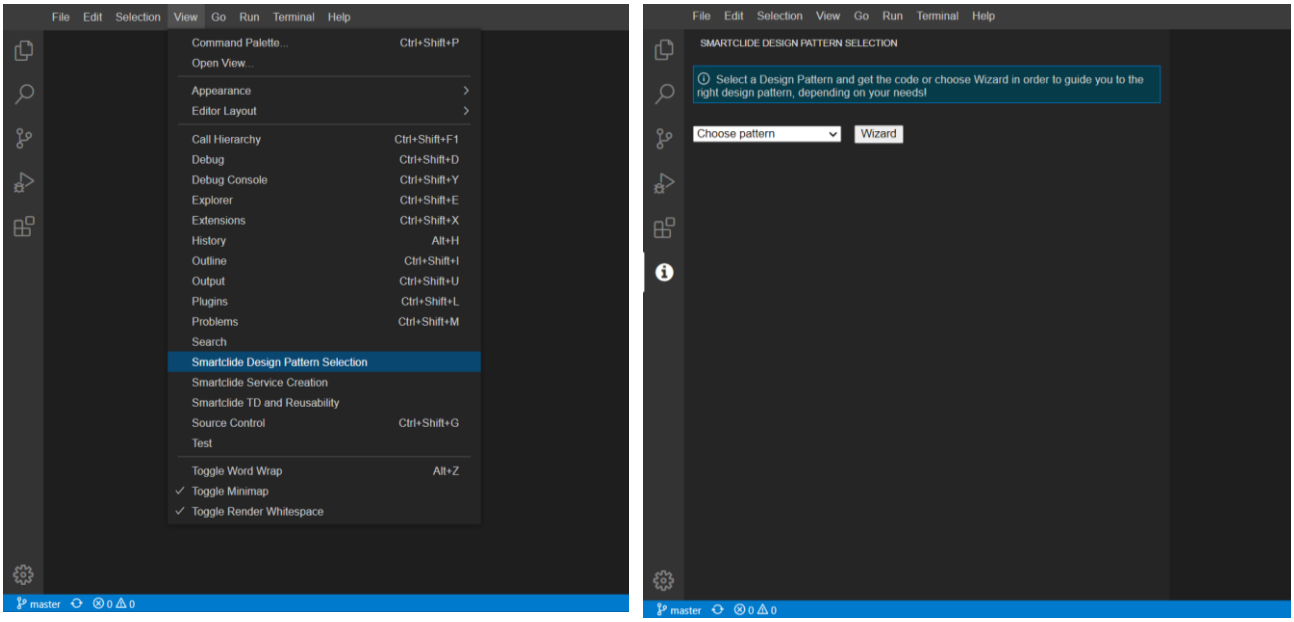

Figure 4. Launching the Theia Extension and Welcome Screen

In the left part of 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 first reminder 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. In the right part of Figure 6, we present the way that the Q&A process of pattern selection appears. For both cases, 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 the top part of Figure 6. In the bottom part of Figure 6, we can see an example of generated code for the `Factory Method` example, presented in Figure 5. 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.

[7] https://hub.docker. com/repository/docker/nikosnikolaidis/theia-td-creation-patterns

[8] http://195.251.210.147:3232/#/home/project/commons-io

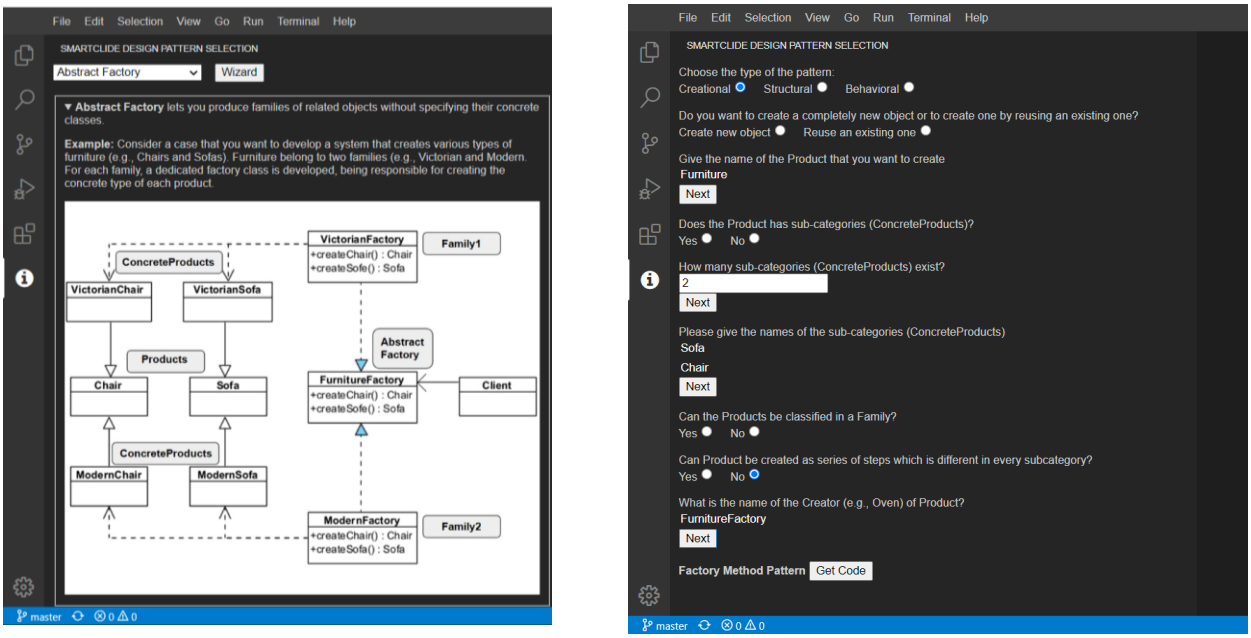

Figure 5. Pattern Selection and Pattern WIZZARD

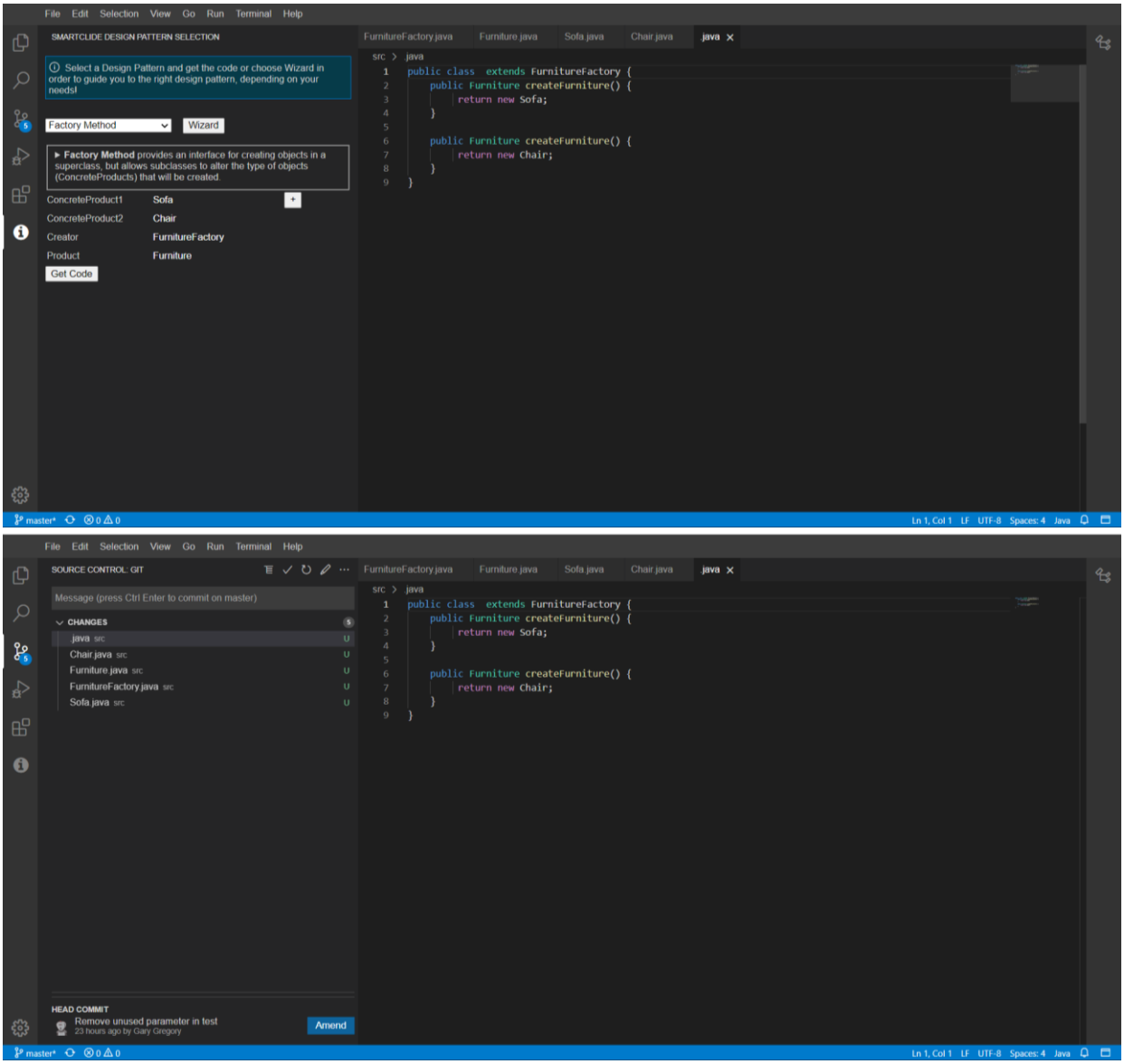

Figure 6. Code Generation

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. [19]. 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 Design Pattern Assistant is to be: (a) relevant to the software industry; (b) aid the correct and timely pattern selection; and (c) usable. 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, timeliness, and usefulness. 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 the correctness, timeliness, and usefulness of each feature (***`EXPERT-MODE`***, ***`WIZARD-MODE`***, ***`CODE-GENERATION`***) has been performed.

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 [19] in the software industry. 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[9] offers full custom service or turnkey package solutions on IT projects. The study is embedded, in the sense that inside the single case (company), 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

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

[9] https://onelity.com/about-us/

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 were assigned a first task, using the ***EXPERT-MODE*** of the Theia Extension (30 minutes); {***Part-D***} The participants were 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); and {***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 C and D). The participants are anonymous and are referred to as `P1-P15`. The distribution of the participants was random, but some constraints were applied: (a) each participant must take one task in the ***EXPERT-MODE*** and one in the ***WIZARD-MODE***; and (b) the same participant cannot be assigned two tasks yielding for the use of the same pattern. 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 RQ_1 was to understand the ***state-of-practice*** in the company regarding ***pattern selection***, and identify the ***benefits*** that can be obtained by using the proposed approach compared to the state-of-practice. In the focus group, we have used four questions related to RQ_1 (see below). Also, data from the pre-study questionnaire have been used, related to experience on patterns and programming experience.

What is your experience with Design Patterns?

How do you choose which Design Patterns to use, or if you will use?

Was the approach and tool helpful? What are the perceived main benefits compared to the state-of-practice?

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. Assume 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.

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 of making them is the same. Moreover, a meal must include items 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.

Bridge Task: Suppose a software system 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 house parts, by selecting the proper animation for each possible pair (e.g., open door, open windows, close roof window, etc.).

Memento Task: In soccer, sometimes after the referee awards a penalty or shows a red card, he/she 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 a system where it will be possible to restore the actions to a previous state.

Upon the participants completing the tasks, the researchers have recorded the values for 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 (based on mapping)

[V7] Confidence Level for Selection (1-5)

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?

Were the questions of the ***`WIZARD-MODE`*** clear? Was there any ambiguity?

Was **Code Generation** useful?

Was it straightforward to **Map Roles to Classes**?

Finally, with respect to the ***usability*** of the Theia extension (RQ_3), 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 quantitative point of view, we relied on the SUS questionnaire [20], which is a state-of-the-art method in the user interface design field. SUS is 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 [20], it allows UI/UX experts to evaluate a wide variety of products and services, including software, mobile, or web applications. The items of evaluation are presented below.

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

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 higher than 68 are considered above average and anything lower than 68 is below average [20].

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 [21]. On the one hand, to obtain ***quantitative results***, we employed descriptive statistics and basic hypothesis testing. For usability, we assessed 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 [22], 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 [23] 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 by both the fourth and the fifth authors, and were discussed and grouped together through an iterative process in several meetings of all authors. The reporting is performed by using codes (frequency table) and participants' quotes. Based on Seaman [21] 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. Findings / Discussion

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 representative quotes and the number of participants referred to 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
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 (RQ_1)

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 (13 out of 15)

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, the majority of novice pattern users (8 out of 10) 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 (7 out of 15) 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 resource 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 (RQ_2)

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 mean 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 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 participates, we added the value that they assigned (1: not useful at all – 5: very useful). To improve the readability of the results in Figure 7, 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 have graded it with 5 points.

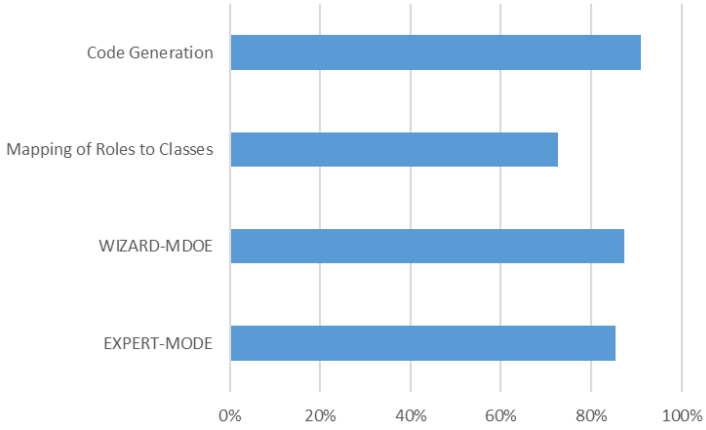

Figure 7. Features Usefulness

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 (4 out of 15) 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 led the participants to the pattern with confidence (DECISION CONFIDENCE—6 out of 15). 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—13 out of 15), 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 (7 out of 15) 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 gave me freedom to pick the pattern that I wanted*!"[10].

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 process. Therefore, not ISOLATING DESIGN FROM CODING (2 out of 15) 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 [24]. Another interesting position was that the integrated code generation will help the developers avoid copying and pasting solutions from the internet, out of which the irrelevant code would need to be removed. The code that the code generation provides is the MINIMUM REQUIRED CODE (discussed from 4 out of 15 participants) 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.

[10] 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`.

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 [25]. Also, some participants (4 out of 15) 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 (3 out of 15 made it explicit) of the implementation, and since the process has a LOW LEARNING CURVE (4 out of 15 made it explicit) 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 (RQ_3)

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.

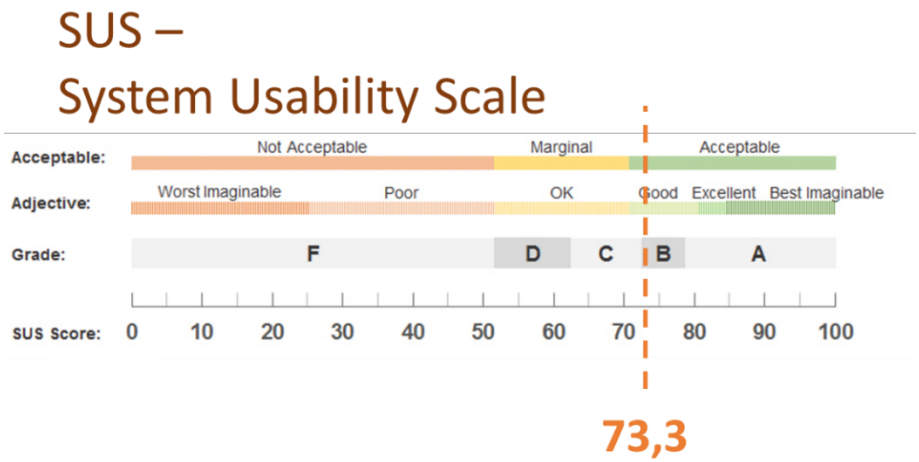

Figure 8. Usability Evaluation Outcome

To study isolated SUS questions, in Figure 9, we present a stacked bar showing the percentage of participants that provided a specific score for each question, based on the SUS questionnaire. We note that for negative answers, we have first calculated the points (inversed the response) and then presented the results—e.g., for statement-1 "*I needed to learn a lot of things before I could get going with this system*" the orange bar corresponds to score ZERO in the original questionnaire. Based on our findings, 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 (blue and red bars) have been received with respect to the LEARNING CURVE and the NEED FOR SUPPORT / MANUAL (long orange and green bars). 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*".

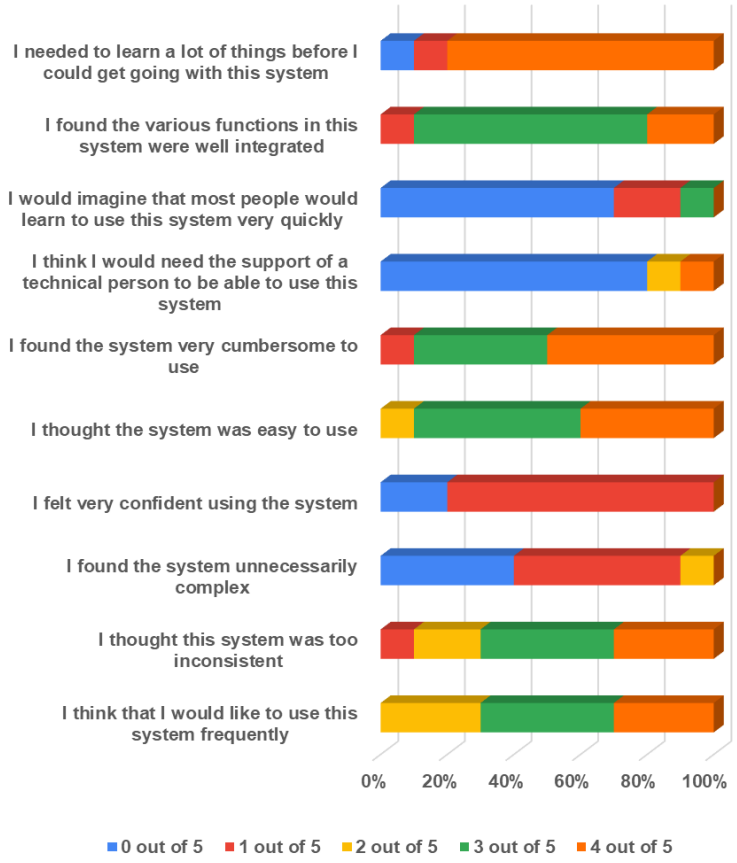

Figure 9. Usability Evaluation Outcome

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. Limitations and Threats to Validity

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 needs (e.g., as it would be achieved by systems such as GitHub Copilot or ChatGPT). Additionally, 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. As a final limitation for this study, we need to note that for systems with large requirements, before the application of the approach, one might need to fragment the requirement to smaller ones; e.g., from an epic to a user study, or from a user story to a task. In that sense, the approach might better fit agile development processes that built the system, upon the development of user stories (where answering the questions of the approach might be more feasible), rather than using pattern selector on existing (already developed) large systems for maintenance purposes. In any case, this approach cannot be used as the only input for design decision making along the development of systems, but it is a useful tool for making decisions during the detailed design of the system.

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 the part of the qualitative evaluation, 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 [26]. 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 such bias cannot be eliminated, method triangulation has been applied to increase the validity of the findings; thereby reducing the corresponding threats [27].

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 save 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, as well as for providing initial implementation of the functional requirement that corresponds to the design pattern. Finally, we believe that a validation of the quality improvement that the pattern solution can bring to the targeted system, would be an interesting extension to this study. A possible positive evaluation, could further boost the applicability of the proposed solution in industrial settings.

Acknowledgement

This work has received funding from the European Union's H2020 research and innovation programme, under grant agreement: 871177 (SmartCLIDE).

References

[1] M. Fowler, "Analysis patterns: Reusable object models", *Addison-Wesley Professional*, October 1996.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, "Pattern-Oriented Software Architecture", *Wiley & Sons*, West Sussex, UK, 1996.

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

[4] B. B. Mayvan, A. Rasoolzadegan, Z. G. Yazdi, "The state of the art on design patterns: A systematic mapping of the literature", *Journal of Systems and Software*, 125, 2017, pp. 93-118,

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

[6] R. C. Martin, “Agile software development: principles, patterns, and practices”, *Prentice Hall PTR*, Upper Saddle River, USA, 2003.

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

[8] J. Bishop, “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, 10-18 May 2008.

[9] B. Keepence and M. Mannion, “Using Patterns to Model Variability in Product Families”, *IEEE Software*, IEEE, 16 (4), pp. 102-108, July 1999.

[10] S. S. Yau and N. Dong, “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 2000

[11] L. C. Briand, Y. Labiche and A. Sauve, “Guiding the Application of Design Patterns Based on UML Models”, *22nd International Conference on Software Maintenance*, IEEE, pp. 234-243, Philadelphia, Pennsylvania, 24-27 September 2006

[12] M. Meyer, “Pattern-based Reengineering of Software Systems”, *13th Working Conference on Reverse Engineering*, pp.305-306, Benevento, Italy, 23-27 October 2006

[13] S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik, S. Bromling and K. Tan, “Generative Design Patterns”, *17th IEEE International Conference on Automated Software Engineering (ASE ‘02)*, pp. 23, Edinburgh, UK, 23-27 September 2002

[14] S. MacDonald, K. Tan, J. Schaeffer and D. Szafron, “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, April 2009.

[15] M. O’ Cinneide and P. Nixon, “Automated software evolution towards design patterns”, *4th International Workshop on Principles of Software Evolution (ICSE’01)*, IEEE, pp.162-165, Vienna, Austria, 12-19 May 2001

[16] N.L. Hsueh, P.H. Chu, P.A. Hsiung, M.J. Chuang, W. Chu, C.H. Chang, C.S. Koong and C.H. Shih, “Supporting Design Enhancement by Pattern-Based Transformation”, *34th Annual Computer Software and Applications Conference (COMPSAC ‘10)*, IEEE, pp. 462 – 467, Seoul, Korea, 19-23 July 2010.

[17] P. Tonella and G. Antoniol, “Object Oriented Design Pattern Inference”, *Journal of Software Maintenance and Evolution*, Wiley, 13 (5), September-October 2001

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

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

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

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

[22] S. Elo and H. Kyngäs, “The qualitative content analysis process”, *Journal of Advanced Nursing*, vol. 62, issue 1, pp. 107-115, 2008.

[23] D. Spencer, “Card Sorting: Designing Usable Categories”, *Rosenfeld Media*, 1st Edition, April 2009.

[24] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, and N. Tsiridis, “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, 2018.

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

[26] Y. Lincoln, and E. G. Guba, “Naturalistic Inquiry”, Newbury Park, CA: *SAGE*, 1985.

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

[28] Falessi, D., Cantone, G. and Kruchten, P., “Do architecture design methods meet architects' needs?”, Working IEEE/IFIP Conference on Software Architecture (WICSA'07) (pp. 5-5). IEEE, January 2007.

[29] Falessi, D., Cantone, G., Kazman, R. and Kruchten, P., 2011. Decision-making techniques for software architecture design: A comparative survey. ACM Computing Surveys (CSUR), 43(4), pp.1-28.

[30] Shahin, M., Liang, P. and Khayyambashi, M.R., 2009, September. Architectural design decision: Existing models and tools. In 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (pp. 293-296). IEEE.

[31] Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H. and Carriere, J., 1998, August. The architecture tradeoff analysis method. 4th IEEE international conference on engineering of complex computer systems (cat. no. 98ex193) (pp. 68-78). IEEE.

[32] Van Vliet, H. and Tang, A., 2016. Decision making in software architecture. Journal of Systems and Software, 117, pp.638-644.

[33] Ionita, M.T., America, P. and Hammer, D.K., 2005, January. A method for strategic scenario-based architecting. In Proceedings of the 38th Annual Hawaii International Conference on System Sciences (pp. 312b-312b). IEEE.

[34] Golfarelli, M., Rizzi, S. and Proli, A., 2006, November. Designing what-if analysis: towards a methodology. In Proceedings of the 9th ACM International Workshop on Data Warehousing and OLAP (pp. 51-58).

[35] B. A. Kitchenham, L., Pickard, and S. L. Pfleeger, "Case studies for method and tool evaluation," IEEE Software”, 12(4), pp. 52-62, July 1995.

[36] C. B. Seaman, “Qualitative methods in empirical studies of software engineering”, IEEE Transactions on Software Engineering, 25(4):557–572, 1999.

[37] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg, “Preliminary guidelines for empirical research in software engineering”, IEEE Transactions on Software Engineering, 28(8):721–734, 2002.

[38] J. M.Verner, J. Sampson, V. Tosic, N. A. Abu Bakar, and B. A. Kitchenham, “Guidelines for Industrially-Based Multiple Case Studies in Software Engineering”, 3rd International Conference on Research Challenges in Information Science, Fez, Morocco, 22-24 April 2009.

[39] C. Wohlin, P., Runeson, M., Höst, M. C., Ohlsson, B., Regnell, and A. Wesslén, “Experimentation in Software Engineering”, Springer, 2012.

[40] Paz, F. & Pow-Sang, J. (2014). Current Trends in Usability Evaluation Methods: A Systematic Review. Proceedings - 7th International Conference on Advanced Software Engineering and Its Applications, ASEA 2014.

[41] Dumas, J. S., & Salzman, M. C. (2006). Usability Assessment Methods. Reviews of Human Factors and Ergonomics, 2(1), 109–140.

[42] Gupta, S. (2015). A Comparative study of Usability Evaluation Methods. International Journal of Computer Trends and Technology. 22. 103-106.

[43] Riihiaho, S, “Usability Testing”. In The Wiley Handbook of Human Computer Interaction, 2018.

[44] L. Pavlič, V. Podgorelec, M. J. C. S. Heričko, and I. Systems, "A question-based design pattern advisement approach," vol. 11, no. 2, pp. 645-664, 2014.

[45] F. Palma, H. Farzin, Y.-G. Guéhéneuc, and N. Moha, "Recommendation system for design patterns in software development: an DPR overview," In 2012 3rd international workshop on Recommendation Systems for Software Engineering (RSSE), 2012, pp. 1-5: IEEE.

[46] R. Rahmati, A. Rasoolzadegan, and D. T. Dehkordy, "An automated method for selecting GoF design patterns," In 2019 9th International Conference on Computer and Knowledge Engineering (ICCKE), 2019, pp. 345-350: IEEE.

[47] A. Naghdipour and S. M. H. Hasheminejad, "Ontology-based design pattern selection," In 2021 26th international Computer Conference, Computer Society of Iran (CSICC), 2021, pp. 1-7: IEEE.

[48] A. Naghdipour, S. M. H. Hasheminejad, and M. R. Keyvanpour, "DPSA: A brief review for design pattern selection approaches," In 2021 26th international Computer Conference, Computer Society of Iran (CSICC), 2021, pp. 1-6: IEEE.

[49] E. M. Sahly and O. M. Sallabi, "Design pattern selection: A solution strategy method," In 2012 international conference on computer systems and industrial informatics, 2012, pp. 1-6: IEEE.

[50] A. Ampatzoglou, S. Charalampidou, and I. Stamelos. 2013. Design pattern alternatives: what to do when a GoF pattern fails. In Proceedings of the 17th Panhellenic Conference on Informatics (PCI '13). Association for Computing Machinery, New York, NY, USA, 122–127

[51] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides and S. T. Halkidis, "Design Pattern Detection Using Similarity Scoring," in IEEE Transactions on Software Engineering, vol. 32, no. 11, pp. 896-909, Nov. 2006