INFORMATICA, 2016, Vol. 27, No. 4, 843–862
© 2016 *Vilnius University*
DOI: http://dx.doi.org/10.15388/Informatica.2016.114

An Entropy-Based Algorithm for Proposing a Suitable Design Pattern

Luka PAVLIČ[1]*, Marjan HERIČKO[1], Vili PODGORELEC[1],
Polona REPOLUSK[2]
[1] *University of Maribor, Faculty of Electrical Engineering and Computer Science Smetanova 17, SI-2000 Maribor, Slovenia*
[2] *University of Maribor, Faculty of Natural Sciences and Mathematics Koroška cesta 160, SI-2000 Maribor, Slovenia*
e-mail: luka.pavlic@um.si

Received: May 2015; accepted: September 2016

Abstract. This paper deals with the problem of selecting a suitable design pattern when necessary. The number of design patterns has been rapidly rising, but management and searching facilities appear to be lagging behind. In this paper we will present a platform, which is used to search for suitable design patterns and for design patterns knowledge exchange. We are introducing a novel design pattern proposing approach: the developer no longer searches for an appropriate design pattern, but rather the intelligent component asks the developer questions. We do not want to invest extra effort in terms of maintaining a special expert system. Guided dialogues consist of independent questions from different sources and authors that are automatically combined. The enabling algorithm and formulas are discussed in detail. This paper also presents our comparison with human-created expert systems via a decision tree. Experiments were executed in order to verify our approach performance. The control group used a human-created expert system, while others were given a proposing component to find appropriate design patterns.

Key words: algorithms, ontologies, design patterns, entropy, selection algorithm, software engineering.

1. Introduction

Design patterns are a proven way to build high-quality software. The number of design patterns has been rapidly rising, but management and search facilities appear to be lagging behind. This is why selecting a suitable design pattern is not always an easy task. It is especially poignant for less-experienced developers. We observed this issue while helping local software development companies to use design patterns wisely and also when teaching students design pattern-related topics. The observation was also clearly stated earlier by different authors: "with more than 20 design patterns in the catalog to choose from, it might be hard to find the one that addresses a particular design problem, especially if the catalog is new and unfamiliar to you" (Gamma *et al.*, 1998). A similar conclusion was

*Corresponding author.

reached by Birukuo *et al.* (2006): “only experienced software engineers who have a deep knowledge of patterns can use them effectively. These developers can recognize generic situations where a pattern can be applied. Inexperienced programmers, even if they have read the pattern books, will always find it hard to decide whether they can reuse a pattern or need to develop a special-purpose solution” (Birukuo *et al.*, 2006).

We developed the Ontology-Based Design Patterns Repository (OBDPR) in order to accelerate design patterns usage. In particular, we addressed functionalities for searching for a suitable design pattern and design pattern knowledge exchange. In this paper we introduce a novel design pattern searching (proposing) approach. The developer no longer searches for an appropriate design pattern, but rather the component on top of the OBDPR asks the developer questions. Based on the answers given, the OBDPR then proposes a design pattern to be used. Dialogues are composed automatically from several independent expert hints.

Using design patterns while developing software is an absolutely necessary measure in the long term in order to keep software quality high. In this context, OBDPR is an important tool, which guarantees that when developers use it, they will solve approximately 50% (see Section 5) of their design problems using an appropriate, already-proven design pattern. This is significantly better than 34%, which can be achieved using traditional help (experience, online resources, books, etc. – see related work, e.g. Pavlič *et al.*, 2014). In addition to this, our approach does not set any additional pressure in terms of maintaining fixed decision trees. Additionally, developers with experience are welcome to include their little part of knowledge in order to help others.

“Hint” in this paper refers to a simple question-answer pairs that give partial information on design pattern usage. An example of an expert hint would be:

What kind of separation do you want in your system?

- by parts that are aware if one (main) part is changed: you can use the “observer” design pattern;
- by parts I can cycle through: you can use the “iterator” design pattern;
- by layers: providing an interface and an implementation: you can use the “bridge”, “proxy”, “facade”, “adapter” or “mediator” design patterns.

This paper will present the approaches, algorithms and mathematics used in order to capture independent expert hints, combine them automatically in a dialogue and in the end, propose design patterns that improve software developers’ performance in terms of correctly selected design patterns. This is why the research is applied to software engineering and it combines concepts from intelligent systems and knowledge management area.

Our work was broken into several steps. Firstly, we proposed a formal ontology-based design patterns representation. Then we implemented a repository with a design-pattern-proposing component. An algorithm for guiding a dialogue, based on separate hints, was developed (an algorithm selects questions dynamically from a set of unasked questions). We also compared the algorithm with a fixed, human-created decision tree (sequence of questions, to be asked in relation to given answers, fixed in the form of a “if-then-else” decision tree by domain experts). Using an experiment, we verified if our algorithm provided good results, when compared to a fixed, human-created decision tree.

To capture those steps, the rest of this paper is organized as follows. In Section 2 we will present some related approaches that tried to cope with the presented issue: how to find appropriate design patterns when needed. In Section 3, the OBDPR platform and its capabilities are briefly presented. Section 4 proposes and describes the algorithm that can be used to combine expert knowledge into dialogues. The algorithm and underlying mathematics is described in detail. Section 5 presents our experiences with the algorithm, built as an intelligent component in OBDPR, and compares the algorithms and formulas with dialogues, based on a human created decision tree. An experiment will also be presented. At the end of the paper, the positive effects of using algorithms, conclusions and possible improvements will be discussed.

2. Related Work

The most closely related work is by Kung *et al.* (2006) where a five-step methodology for constructing an expert system was presented that subsequently suggests design patterns to solve a design problem. The focus is on the collection and analysis of knowledge on patterns in order to formulate questions, threshold values and rules to be used by an expert shell. A prototype of the Rule-Based tool (for a subset of GoF patterns) was developed that selects the design pattern through dialogue with the software designer to narrow down the choices. The evaluation will also be presented in this paper. The focus is on verifying if our entropy-based algorithm enables developers to find a suitable design pattern on the basis of little costs (i.e. without constructing a decision tree, using a special expert system). Our research is not aimed at forming dialogues (generating questions and answers – although the approach does allow the further inclusion of machine-generated questions). Instead, we primarily allow experts to give independent design hints that are automatically and dynamically linked to dialogues.

Gomes *et al.* (2003) introduced an approach that is based on Case-Based Reasoning (CBR) and WordNet. This approach is based on an idea that a system can learn to select and apply design patterns if it can store and reuse an experience that reveals the situation in which patterns are used. The application of a specific pattern to a specific software design is represented in the form of a design pattern application (DPA) case. A DPA case describes a specific situation where a software design pattern has been applied to a class diagram. DPA cases are stored in a case repository and indexed using a general ontology (WordNet). A target class diagram is used as the problem description for which a search for similar cases in the repository is carried out. Our approach differs in the aim that we only wish to assist a developer in selecting a suitable design pattern and there is no need or demand for having a context already described as a UML class diagram. Our approach does not require a developer to prepare a model first. The information needed to identify a suitable solution is gathered using a guided dialogue, where a developer provides information on problem characteristics by choosing items from the lists of available answers. This technique of gathering context is also different – we use guided dialogues.

Dietrich and Elgar (2005) introduced one of the possible formal design patterns description using ontologies in standard notation, namely OWL. Their main focus was on creating notation for a detailed description on pattern structure (with dedicated OWL classes, e.g. ClassTemplate, MethodTemplate, etc.). The ontology ODOL (Object Design Ontology) is primarily used in order to recognize design patterns that are used in existing solutions with the Design Pattern Scanner tool. This work is complementary to ours, since we do not focus on design pattern structure, but rather aim to capture knowledge of design patterns. However, in our ontology, we link our patterns to ODOL. Therefore, our approach could be used in order to find suitable design patterns while an ODOL ontology could further be used in assistance when implementing the selected design pattern.

Kampffmeyer and Zschaler (2007) are authors that employed a similar approach to ours. They also use ontologies as a tool for the formal description of design patterns with a similar aim: finding appropriate design patterns in a given situation. They created an ontology, called "Design Pattern Intent Ontology", which captures the intent of a given pattern and a set of problems that are solved by a given pattern. Based on the ontology, the authors prepared a proof-of-concept tool, called "The Design Pattern Wizard". It allows user to select several problem situations, and the tool returns all design patterns, related to a given set of problem situations. Our approach enables us to collect expert-given hints on design pattern usage, while the presented tool then automatically guides a short dialogue with user in order to select a suitable design pattern. Our ontology is simpler and focused more on short, straightforward hints and their smart usage via a novel algorithm for guiding a dialogue.

In the paper "A Question-Based Design Pattern Advisement Approach" (Pavlič *et al.*, 2014) the authors present an ontology and question-based advisement (OQBA) approach. In the paper, ontologies are selected as a knowledge formalization technique. Several aspects, including design pattern structure and behaviour, are formalized in order to support DPEX (Design Patterns Expert) tool. The main idea of the approach is similar to the one presented in this paper: asking developers questions in order to propose appropriate design patterns in a given context. The paper proves that such approach significantly improves design pattern adoption in terms of selection correctness and time spent for searching. It also proves that such approach not only helps novice developers, but also experienced ones. Like the OQBA paper, we follow the same philosophy: asking developers questions in order to propose a solution. However, the research presented in this paper, goes beyond the findings of the OQBA paper. We introduce flexibility in terms of defining questions and answers (we use the term "hint") that are used during the guided dialogue. Question-answer pairs are not connected by design pattern experts, but instead use our algorithms in order to guide a dialogue. This is how we eliminate high domain expert involvement in terms of defining decision systems, maintaining a knowledge base, etc. And most importantly: we would like to build a simpler system in terms of performance (the ability to propose a correct design pattern), close to human-generated dialogues (decision trees), guided by if-then-else decision trees. In order to do so, we have prepared different foundation – algorithm approaches, based on game theory and information entropy theory with a mathematically sound set of formulas in order to select and combine independent questions, etc.

This research is also based on the papers Pavlič and Heričko (2008) and Pavlič *et al.* (2009). In the paper "Semantic Web as an Enabling Technology for Better Design Pattern Adoption", the authors present the OBDPR platform, which has full semantic web potentials in terms of defining an ontology and combining design pattern knowledge. The platform is used in order to experiment with different searching possibilities, including guided dialogues (random dialogues, fixed dialogues, etc.). The entropy-based algorithm, presented in this paper, is implemented on top of the OBDPR platform. Secondly, in the paper "Improving Design Pattern Adoption with an Ontology-Based Repository" the authors focus on formalizing design patterns descriptions and selecting an appropriate approach for this challenge. A set of dominant techniques (including DPML, RSL, ODOL, LePUS, Slam-SL and others) is reviewed and discussed. The OWL-based formalization technique has been recognized and selected as the most promising knowledge formalization technique in the area of design patterns.

As will be discussed in more detail later, Shannon's information entropy from information theory (Shannon, 1948) was used in our platform. Other authors have also used it. For example, in the paper "Question Answering Using Maximum Entropy Components" (Ittycheriah *et al.*, 2001) the authors use knowledge taken from an encyclopedia to find answers for users' questions. They use information entropy as a measure for answer relevance. We use entropy as a measure of information given the context of a problem. Questions and answers are used as a tool for lowering the current information entropy on problem context, until an (experimentally determined) threshold value is reached.

3. The Platform: The Ontology Based Design Pattern Repository

The OBDPR is a platform and an application for exchanging knowledge about design patterns. Among others, it enables simplified searching. It has a web-based user interface. The OBDPR is not only a design pattern repository. It is ontology-based, which allows for it to be a platform for building intelligent services.

The OBDPR is used as a proof-of-concept platform and tool within the research, presented in this paper. It was used as a design pattern repository, design pattern editor, question-answer pair editor and platform, used during the experiment.

As such it includes several functionalities:

- Holds a repository of design pattern containers and design patterns.
- Integrates knowledge on a particular design pattern from the web and additional data sources.
- Allows design-pattern experts to annotate patterns with additional knowledge.
- Enables users to view the repository in plain RDF/OWL form with the possibility of executing predefined/advanced SPARQL queries.
- Transforms current RDF data to provide a user-friendly view on design patterns.
- Provides questions and answers for a particular design pattern with the means of a FAQ (Frequently Asked Questions), which allows design-pattern experts to transfer their implicit knowledge to less experienced software engineers.
- Indexes all the integrated data for supporting full text search capabilities of services built on the platform.

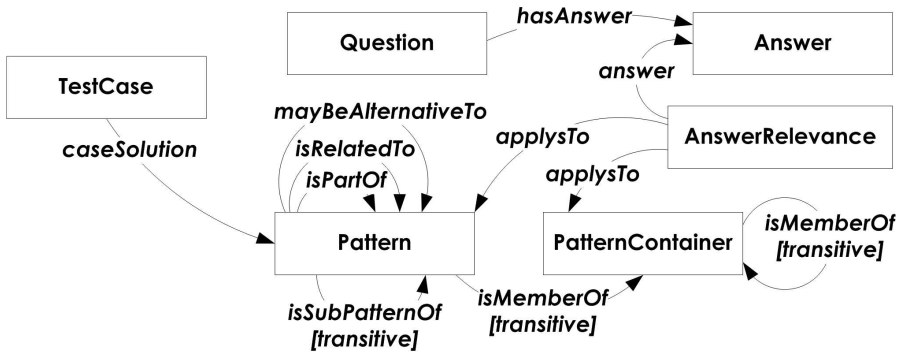

Fig. 1. OBDPR ontology highlights.

- Allows full access to RDF data to services built on the platform, including questions and answers, which enable intelligent services to use an expert system-like proposing or validating of services.
- Includes a set of real-world examples and appropriate design patterns solutions in order to enable services to be used to train users or to demonstrate the appropriate use of design patterns in real-world examples.

There are also three intelligent services running on top of the platform: a full-text search service, a proposing service to support design pattern selection and a training service using real-world examples from an underlying ontology. The OBDPR prototype includes all design patterns found in GoF and J2EE design pattern catalogues. It is also open for other sets of design patterns; inserting additional patterns is a straight-forward operation. GoF (Gamma *et al.*, 1998) and J2EE (Core J2EE Patterns, 2013) catalogues give us enough opportunities to perform relevant experiments with design patterns in real-world examples and with real developers. As mentioned earlier, even repositories of that size have been shown to be problematic (Gamma *et al.*, 1998).

The OBDPR underlying ontology is implemented using OWL (Web Ontology Language, Recommendation by W3C) (W3C Semantic Web, 2013). A highlight ontology fragment is shown in Fig. 1 (a detailed set of attributes is not shown). The concepts “Pattern” and “PatternContainer” are used to capture knowledge on design patterns. Design patterns are grouped into Pattern Containers. In this context, they represent sets of design patterns (e.g. behavioural, structural design patterns, etc.). We use a hierarchical organization of pattern containers. Every pattern container may contain several pattern containers and patterns. This enables us to capture several divisions of design patterns, not only those found in fundamental literature (for example GoF patterns are usually divided into three groups – structural, behavioural and creational design patterns – but we can also find other divisions of the same patterns). Every pattern can be included in several containers; the same is true for containers. This happens even if multi-container membership is not explicitly written down, since the “isMemberOf” relation is declared to be transitive. Patterns themselves are connected in a more logical way by means of related, similar, composed patterns and pattern hierarchies. This information is used when we propose a design pattern: the user can browse similar, related or complementary design patterns in addition to examining the only proposed design pattern. Not only are patterns and pattern containers themselves included in the ontology, but there are also real-world examples using patterns to give more meaning to the OBDPR user ("TestCase" class).

The core of the ontology (Pattern, PatternContainer) is the same as the one, established in related work of Pavlič *et al.* (2014). This is why expert knowledge (hints in terms of questions and answers) is modelled in a manner that is both user-friendly and computable. Design pattern experts can provide experiences (hints) in question-answer pairs. This enables them to capture their implicit knowledge in design patterns. Not only can experts utilize experiences to tell which design pattern should be used in a particular real-life situation ("Question" class), but they can also specify more possible solutions to real-life situations ("Answer") with a specified probability ("AnswerRelevance") – see Fig. 1. This is a value from [0, 1] and tells a user how likely it is that a particular candidate ("Pattern" or "PatternContainer") is used when the answer to a given question is confirmed as positive. It basically tells us what the certainty is that a particular candidate is useful if the answer is selected:

- value 0 means it is certainly not likely that the candidate is a final solution;
- value 1 means it is certainly likely that the candidate is a final solution;
- values between 0 and 1 tells us how likely it is that a particular candidate is relevant as a final solution, i.e. 0.5 means we have no information whatsoever if the candidate is relevant or not, 0.25 would mean it is not very likely that the candidate is the solution, while 0.75 would mean it is quite likely that the pattern is the proper solution.

Questions, their possible answers and the connected candidates' relevance are collected by design pattern experts. The hint repository can change all the time, since experts are given the opportunity to insert, edit, or even delete questions and answers on a daily basis using the platform's user friendly interfaces.

One example of an expert hint would be:

What kind of separation do you want in your system?

- by layers: providing an interface and an implementation $\longrightarrow$ Proxy: 0.7, Facade: 0.7, Bridge: 0.7, Adapter: 0.7, Mediator: 0.7;
- by parts I can cycle through $\longrightarrow$ Iterator: 1.0;
- by parts that are aware if one (main) part is changed $\longrightarrow$ Observer: 1.0.

The numbers shown above are subjective assessments, given by domain experts. For candidates (Patterns or Pattern Containers) that are not mentioned in hints, we assume a value of 0.5. The presented hint example covers just a few design patterns, others are left in a "do-not-know"-zone (value 0.5). It could happen that several experts would give rather different (subjective) values to the same hint. This is why trust-related concerns could emerge. However, as will be demonstrated later in the paper, our algorithm automatically deals with issues when the given relevance values could be contradictory. Having contradictory hints in the repository only means a longer dialogue, not wrong outcome.

4. Algorithm Driven Dialogue

In the previous section, we presented the idea of formally capturing knowledge about using design patterns, e.g. if an inexperienced developer has a problem using existing classes in a markedly different way, the expert would say: "Do you only need to use it, or do you want the interface to be simplified at the same time? In the first case I would propose using the Adapter pattern, while in the latter case it would be good to use the Facade pattern". This very informal means of communication is used on a daily basis. And it is also supported by our platform. There are not many situations where one can find an appropriate solution with only one hint. In order to connect several hints (question-answer pairs) into a dialogue, an oft-used technique in expert systems is to connect them into a decision tree, which can be used for guiding dialogues in pursuing a final solution. We believe that such an approach is a) quite expensive, since you need a lot of work from your domain experts b) inappropriate in the long-term, since updating complex decision trees with new questions is not an easy task and c) impractical, since experts would not be particularly motivated to connect questions to form a decision tree, whereas it is not as taxing to just provide a hint or two.

In this section we will present our algorithm, which enables us to connect those hints into guided dialogues.

4.1. *The Goal*

As previously mentioned, the main objective of our algorithm is to dynamically connect separate hints into a guided dialogue as an alternative to constructing fixed decision trees. There could, however, be several techniques on how to select an appropriate solution, based on answers given to the posed questions. One could simply ask all possible questions and use simple mathematics to select a design pattern that was given the highest relevance through such a dialogue. A second technique would be to ask random questions for as long as some kind of measure could not be found that a particular pattern candidate had significantly greater value than others. We decided to guide a dialogue more cleverly: after the user answers a question, we would choose the next most promising question. We wanted to ask as few questions as possible. In an ideal scenario, we would ask as few questions as would be asked if we would have a fixed, domain experts-created decision tree. The main idea of the algorithm is therefore to select such a question at every step, which would maximally increase the captured knowledge about the current design issue (see Section 4.4).

Our algorithm has three possible termination states: one candidate is chosen based on the predefined threshold, the user terminates the algorithm manually, or there are no more questions to be asked.

4.2. *Input and Output Data*

The input to the algorithm would be a set of all hints (question-answer-relevance structures) and a set of all candidates in OBDPR. The algorithm would dynamically choose questions from the set in order to give an output: a set of possible design patterns. In ideal conditions, in the output set there would be only one outstanding appropriate design pattern.

Let us present a set of candidates where the vector $P = [p_1, p_2, p_3, \dots, p_n]$, where every candidate (pattern) is presented as p_i, where $1 \leqslant i \leqslant n$. The number of all possible candidates is n.

In the algorithm itself, and as an output, the vector C is used. Its structure is as follows: $C = [c(p_1), c(p_2), c(p_3), \dots, c(p_n)]$. Let us call vector C a "certainty vector", where $c(p_i)$ is the current certainty for pattern p_i and $c(p_i) \in [0, 1]$. Its meaning is very similar to the relevance in answers, as discussed in Section 3:

- the number 0 means that it is certainly not likely that the pattern is the final solution;
- 1 means it is certainly likely that the pattern is the final solution;
- while with 0.5 we have no information about the pattern whatsoever.

It is obvious that the initial value of vector C is $c(p_i) = 0.5$ for all i. It is the goal of the algorithm to employ questions until vector C is transformed as much as possible.

When vector C is evaluated with a value that is deemed good enough (at the moment it is the experimental determined threshold), the algorithm is finished. If this is not the case, the next most promising question is selected. In subsequent sections, we will show how to select certain questions from a set of all unasked questions and how we propose calculating changes in vector C, based on the answers given.

4.3. *Combining Answers for the Current Certainty Vector*

When the answer is given during the dialogue, vector $A = [a(p_1), a(p_2), a(p_3), \dots, a(p_n)]$ is constructed. The values $a(p_i)$ are the values of relevance for a particular candidate in the given answer (see the example in Section 4). For candidates that are not mentioned by answer, we use the value 0.5 (which means that we have no information on the candidate). When changing vector C (current certainty vector) with regard to vector A (the current answer), several guiding factors should be followed:

- Values $c(p_i)$ should remain within the interval [0, 1].
- If $a(p_i) = 0.5$, the value of $c(p_i)$ should not change, since we have no additional information on the candidates' relevance.
- More than $a(p_i)$ is higher/lower than $c(p_i)$, more the value $c(p_i)$ should rise/fall. But not to $a(p_i)$ level, since we do not want one (last) answer to define the final state of the value $c(p_i)$. There will typically be more answers given regarding the same candidate.
- The more answers that are given, the less they should impact vector C. This is how we want to regulate the questioning: rough at the beginning but after getting more answers, we want to refine profile vector C in a more precise way.

Figure 2 shows one possible scenario: $c(p_i)$ should change to $c_{new}(p_i)$ only in cases where $a(p_i)$ is different from 0.5. Their impact ($c_{new}(p_i)$) should be somewhere between $c(p_i)$ and $a(p_i)$.

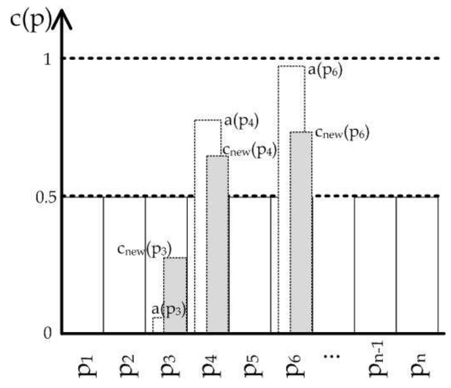

Fig. 2. Combining the certainty vector with the answer vector.

One of the possibilities was the method of average value (see equation (1)).

$$c_{new}(p_i) = \begin{cases} \frac{c(p_i)+a(p_i)}{2}, & a(p_i) \neq 0.5, \\ c(p_i), & a(p_i) = 0.5. \end{cases} \tag{1}$$

This simple method meets all the guidance factors mentioned above. In addition, it is also resistant to contradictory answering: if the developer provides an answer in favour of a particular candidate and later an answer that is not in favour of the same candidate, we are again close to 0.5 (the "do not know"-zone).

In addition to eliminating contradictory answers during the dialogue, this method also ensures the quality of the question-answer pair-based knowledge base. Since our method enables a lot of knowledge to be inserted without a central quality-check point, this property is important. In cases where domain experts include contradictory hints for the knowledge base, this would just prolong the dialogue, so possible contradictory hints would eliminate each other.

4.4. *Evaluating the Current Dialogue State*

The evaluation of the current dialogue state (vector C) is used to:

(a) determine if there is one candidate that can be significantly chosen as a final solution and
(b) select the next question to be asked in the hint repository.

In the first case, we calculated the evaluation value of vector C, and if the value happens to be lower than the predefined threshold (since we base our approach on entropy theory, a lower value is better) then the dialogue can stop. When choosing the next question, we calculate the values of vector C with all possible answer vectors A. The algorithm selects those questions that definitely and maximally lower the value of vector C in every case.

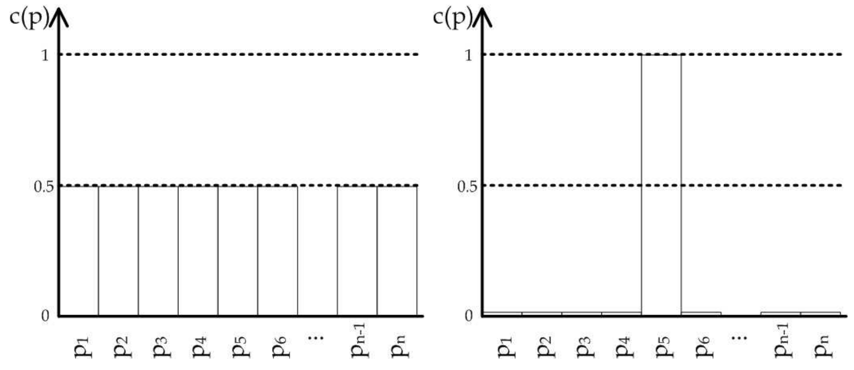

Fig. 3. The worst and best situations.

Let us briefly discuss what the evaluated value of vector C would be:

- If all patterns share the same certainty, ($c(p_i)$ is the same for all i) the value should be maximal, since such a vector does not provide any information about which pattern to propose whatsoever (see: Fig. 3).
- The value should be minimal when one pattern has certainty 1 and all others share certainty 0. In this case, it is not possible to add more information needed to propose a pattern (see Fig. 3; it is trivial to guess which candidate is the winner).
- The more pattern certainties that share the value 0.5 (the "do not know"-zone), the larger the value should be.
- Less patterns standing out from the average certainty value means a smaller evaluated value. E.g. if one pattern has certainty 0.8 and all others 0.4, the value should be lower than if two certainties are close to 0.8 and all others are 0.4. The first possibility gives us more information about which pattern to propose, while the second one shows doubt about which of the two candidates to select.

We designed the algorithm in such a way that the evaluation method is separated from the algorithm itself. The most promising was the formula to calculate information entropy from vector C, as defined by Shannon (1948). The process of the dialogue would then basically be the process of lowering the entropy value: when the developer starts the dialogue with the platform, the entropy value is maximal, since we have no information regarding the proposing of the design pattern. Theoretically, if the user answers all the questions held by the platform, the entropy would be reduced to the minimum possible level and it would be quite simple to propose a pattern while counting answers in favour of a particular pattern. The formula, as defined by Shannon (1948), in the case of vector C would be:

$$Ent(C) = -\sum_{i=1}^{n} c(p_i) \cdot \log_2 \left(c(p_i)\right). \tag{2}$$

One could argue that entropy is not limited by a maximum value, as defined by thermodynamics and by Shannon in information theory (Shannon, 1948). However, our particular problem has well-known less and most entropic states (see Fig. 3), so the entropy is limited in our case.

The formula, based on entropy, worked well and was shown to be a good guide for selecting a question to ask. However, we observed issues when two different vectors C were evaluated as the same, even though they contained rather different information. This prompted us to ask more questions and have longer dialogues as a result. The reason for this was quite clear: the Shannon's entropy formula does not capture relationships between individual elements, it only evaluates elements on their own and summarizes the partial values. How to define the threshold value was also an issue: equation (2) gives entropy values, which are dependent on the size of vector C (with a higher n value, higher entropy values are to be expected). This is why we developed our own formula, which works on our domain even better than the original general formula for entropy. However, all Shannon's original ideas about entropy (continuity, symmetry, maximum, additivity) and lowering entropy are preserved (see Shannon, 1948).

Let us begin with some definitions: let $c_{\max} = \max\{c_i \mid 1 \leqslant i \leqslant n\}$ and let $count(c_{\max})$ denote the number of pattern certainties, sharing the maximum value in vector C that is the number of the most promising candidates. Similarly, let $count(c_{05}) = |\{c_i \mid c_i = 0.5 \text{ and } 1 \leqslant i \leqslant n\}|$ that is the number of pattern certainties, sharing the value 0.5 (the "do not know" zone). Moreover, let $p_{05} = \frac{count(c_{05})}{n}$ and $p_{\max} = \frac{count(c_{\max})}{n}$ that is p_{05} represents percentage of those patterns that have the value 0.5 and $p_{\max}$ percentage of those patterns that share the maximum value of vector C. Finally, let $\overline{c}$ be the average value of vector C calculated as

$$\overline{c} = \frac{1}{n}\sum_{i=1}^{n} c(p_i).$$

Then we calculate the relevancy of vector C in the following way:

$$Rel(C) = 1 - (c_{\max} - \overline{c}) \cdot D \tag{3}$$

where

$$D = 1 - \alpha_1 \cdot p_{05} - \alpha_2 \cdot p_{\max} \tag{4}$$

and α_1 and α_2 represent certain positive weights that must be applied to each factor. It must hold that $\alpha_1 + \alpha_2 = 1$.

In the formula of $Rel(C)$ factor D plays an important role in a way of controlling the procedure of asking questions. In the beginning, we want to ask general questions and later on we begin by asking certain questions based on the previous answers. More precisely, the main idea of factor D was to incorporate the percentage of all design patterns, involved in the dialogue (n subtracted by the affected candidates, then divided by n). In the beginning, factor D is close to 0, but while a dialogue progresses, factor D more closely approaches 1. The testing has shown that the formula works best if we give some weight to factors p_{05} and $p_{\max}$. In our case, we have weight $\alpha_1 = 0.75$ in favour of having candidates addressed by even one question and it is 25% important that we have few candidates that significantly step out of the average, therefore $\alpha_2 = 0.25$. With trivial mathematical derivation, we end up with the formula shown in equation (4). One might also choose different values for α_1 and α_2 in which $\alpha_1 + \alpha_2 = 1$ ($\alpha_1 = 0.6$ and $\alpha_2 = 0.4$, $\alpha_1 = 0.7$ and $\alpha_2 = 0.3$, etc.) for cases where one would like to find the best combination. However, it was not the aim of our research to prove what the best working ratios were.

Summing everything up we end up with the formula of D, the value of which lies in $[0, 1]$: obviously $p_{05}, p_{\max} \in [0, 1]$, therefore $\alpha_1 \cdot p_{05} \in [0, \alpha_1]$ and $\alpha_2 \cdot p_{\max} \in [0, \alpha_2]$. Moreover, $\alpha_1 \cdot p_{05} + \alpha_2 \cdot p_{\max} \in [0, \alpha_1 + \alpha_2] = [0, 1]$ which leads to $D \in [0, 1]$.

Now let us return to equation (3): observe that also $c_{\max} - \overline{c} \in [0, 1]$, therefore $Rel(C) \in [0, 1]$. If $c_{\max} - \overline{c}$ is close to 1 (this is in case when all but some values of c_i are close to 0 and one close to 1 (the case we want to end up with), the value of $Rel(C)$ will be close to 0, which is the desired entropy. In the case of the vector C that we start with (that is $c_i = 0.5$ for all i), the value of $Rel(C) = 1$ (because $c_{\max} = \overline{c} = 0.5$), which is the highest entropy – we do not have any information yet.

4.5. *The Algorithm*

In this section, we will present the whole algorithm, as discussed throughout this paper. The function "guide_dialog" is the primary one. It is used to select, ask questions and collect answers until its termination because of a) the user b) a solution has been selected or c) there are no more questions to be asked. Since the user observes the progress of vector C all the time (visual element in OBDPR), it is easy to see when a particular candidate is starting to gain. The user is also given the opportunity to skip questions that he or she does

```
 1: create a set of all questions QUE
 2: create a set of all candidates P
 3: procedure GUIDE_DIALOG(QUE, P)
 4:     THRESHOLD = 40.45
 5:     create vector $C$, $c_i = 0.5$, $1 \leqslant i \leqslant n$, $n$ = candidates number
 6:     while QUE not empty do
 7:         Q = select_question(QUE,C)
 8:         if Q == empty then
 9:             print $C$, terminate
10:         end if
11:         remove Q from QUE
12:         collect answer ANS
13:         if ANS! ="do not know" then
14:             create vector V_ANS from answer ANS
15:             $C$ = combine($C$, $V\_ANS$), print $C$
16:             if (user select to finish) or ($Rel(C)$ < THRESHOLD) then
17:                 terminate
18:             end if
19:         end if
20:     end while
21: end procedure
```

```
# NQUE is a set of all unasked questions
# TS is a current state of certainty vector
function SELECT_QUESTION(NQUE, TC) → QUESTION
    create vector MIN; MIN[$i$] = 1, $1 \leqslant i \leqslant n$
    current_rel = $Rel(TC)$
    # verify all possible answers, find max change in TC,
    # even if worse answer is given
    for for every question in NQUE do
        min_difference = 1
        for every possible answer do
            create vector A from answer
            tempC = combine(TC,A)
            created_diff = current_rel − Rel(tempC)
            if created_diff < min_difference then
                min_difference = created_diff
            end if
        end for
        MIN[current question] = min_difference
    end for
    find index $i$ of maximal value in vector MIN
    if MIN[$i$] < 0 then
        return no question # no answer can increase the knowledge
    end if
    return NQUE[$i$]
end function
```

```
# $C$ is current state of certainty vector
function $Rel(C) \rightarrow Rel$
    calculate progress factor from $C$ as $D = 1 - \alpha_1 \cdot p_{05} - \alpha_2 \cdot p_{\max}$
    calculate relevancy of $C$ as $Rel(C) = 1 - (c_{\max} - \overline{c}) \cdot D$
    return Rel
end function
```

```
# $C$ is certainty vector
# $A$ is answer to combine with
function COMBINE($C$, $A$) → $c_{new}$
    create vector CN; $CN(p_i) = c(p_i)$
    change values in CN as $CN(p_i) = \frac{c(p_i) + a(p_i)}{2}$ if $a(p_i) \neq 0.5$
    return CN
end function
```

not understand or does not know the answer to. The threshold value of 0.45 for $Rel(C)$ was experimentally determined (meaning that during simulations and real experiments, lower values did not improve the quality of the proposed patterns; they only prolonged the dialogue). This value can, however, be easily changed in order to investigate its ideal value.

The function "select_question" receives a list of all unasked questions and the current state of vector C. The output is a question to be asked, if possible.

The function "rel" calculates the relevance value of vector C as discussed in Section 4.4.

The function "combine" accepts vector C and it combines with a vector that is created when the answer is given. This function is also used when algorithms combine all questions with the current vector C in order to determine the best question to ask.

5. The Experiment

With this experiment, we wanted to test the hypothesis: entropy-based algorithm for guiding a dialogue is not significantly different (in terms of correctness) than a fixed, human-created decision tree. The type of guiding the dialogue was an independent variable. We measured developer decision correctness and the length of dialogues.

Input data for experimental purposes was acquired by domain experts (experienced senior developers and architects from partner companies and researchers with significant design pattern knowledge). We ended up with 36 hints, dealing with 32 different design patterns. Every hint was connected with an average of 2.7 candidates (minimal 2, maximal 4), which implies that these hints were real-world simple independent hints – not artificial, perfectly polished ones. One could argue that the presented numbers are low. However, it was our idea that volumes were something that we had to live with on a daily basis as a developers, since hints came out of a real environment. When taking into account the presented formulas and algorithms, if our approach worked well with volume of that size, then there is no objective obstacle to it not working even better when a larger knowledge base is available.

For the purposes of comparison, we also asked domain experts to construct a competing expert system with a fixed decision tree (using the same hints, not new ones). The same hints from the repository were fixed in a simple decision tree, with simple if-then-else rules that connects questions and answers to the final proposal. Basically, the fixed dialogues with fixed outcomes were created in an opposition to our dynamic, algorithm driven dialogues. With such a created fixed tree, it was possible to guide 72 different dialogues, 3 of which gave no solution to the user. We believe that such a tree can guide dialogues the best, since it was completely constructed and verified by domain experts from the industry and academy. It was our aim to figure out how well our algorithm performs, compared to a human-, hand-constructed tree.

We invited 40 candidates to be involved in the experiment. They were all given 19 simple cases (please find typical one later) where design patterns might be used. 17 of the candidates were developers from industry with several years of experiences in software development, 23 of them were final-year IT students with a solid knowledge of object-oriented systems, software development, architectures and best practices. They were all asked to solve as many examples as possible. Half of them (industry people and students, mixed) were solving cases using dialogues, guided by a fixed decision tree, while half were using our algorithm. We measured how many questions were answered in every dialogue and if they managed to find an appropriate design pattern.

Typical case during the experiment:
You are developing a PhoneBook class. It will enable you to manage your contacts in your mobile phone. A PhoneBook class should reuse already existing classes that enable the easy use of mobile phones, for example calling contacts or sending an SMS to a contact. At development time you do not know what the classes should look like, so you use your own dummy-class MyPhone. After getting the class AndroidPhone you notice that MyPhone and AndroidPhone are similar but have different methods covering the same functionality. You want to use the AndroidPhone class, but are not willing to change your classes and keep using MyPhone to use AndroidPhone functionalities.

Which design pattern would you use to solve the issue?

The results are as follows:

Candidates using a fixed decision tree were guided with dialogues containing 3–7 questions. The average dialogue had 4.3 questions. They managed to find 48% (standard deviation 21%) of the right solutions.

Candidates using our algorithm were asked to answer 7–10 questions in each dialogue (average 8.5), and they answered an average of 1.2 questions in a dialogue with "do not know" (maximum 3). They managed to find 45% (standard deviation 32%) of the right solutions.

A typical recorded dialogue, guided by our algorithm:

System: Do you want to change the way that existing functionalities are used in your class?
User: No.
System: What behavioural aspects do you want to change?
User: The way the objects communicate.
System: What do you want to do?
User: Define or change class architecture in my system.
System: How do you want to connect to the interface of the existing class?
User: I will use my own interface that is different from the existing one.
System: Do you have many methods to choose from in the runtime?
User: No.
System: Do the objects share the same behaviour?
User: Yes.
System: Do you want to have the possibility of returning to the previous object state?
User: No.
System: Do you want a collection of objects to react to a change in a single object?
User: No.
System: You should use pattern Adapter.

The results, given by the dialogue, guided by fixed, expert-created decision tree and those, guided by entropy-based algorithm have proven not to have a statistically significant difference. In previous research (Pavlič *et al.*, 2014) we showed that the question-based approach helps in terms of design pattern identification correctness. This is why the outcome is not surprising. However, developers had to answer more questions using our algorithm guided dialogue, but on the other hand, our approach means no additional effort for domain experts (e.g. constructing a fixed dialogue or changing the existing with new question-answer pairs). We also see a lot of potential in integrating several knowledge bases of separate hints and dialogues, having all hints incorporated would then be a logical consequence.

The correctness rate (45% and 48%) might not sound like a lot in terms of finding a correct design pattern while designing software. However, it should be noted that this does not mean that developers are designing their software incorrectly in more than 50% of cases. It means that they are not using the appropriate, already-proven design patterns for solving design problems. Instead of this, they are inventing solutions on their own. One could say that in over 50% of cases, software architects are reinventing the wheel. The correctness rate, achieved in this experiment, is better than conventional help (i.e. books, online resources, experiences) – see Pavlič *et al.*, 2014 by a factor of 1.48.

6. Conclusions

The experiments showed that using our algorithm did not mean that the results, in terms of correctness, would be different if domain experts invested a lot of effort to create a fixed decision tree. The dialogue length was also satisfactory. With 36 hints, and 32 patterns, less than a quarter of the chosen separate hints would give one enough information in order to decide on which design pattern to include. Having more hints would lower that number even further.

The experiment shows that candidates, using a fixed, human-created decision tree, were guided by dialogues with an average length of 4.3 questions. They managed to find 48% of the right solutions (aligned with already proven and published design patterns). Candidates using our algorithm were asked 8.5 questions on average per case. Correctness rate was 45% which is not statistically different.

Our approach provides the same performance without high domain expert involvement. However, the dialogue length is rather longer. Having the possibility to integrate hints from several domain experts, we could say that our algorithm enables a straightforward knowledge transfer from experienced developers to inexperienced ones. Such a knowledge transfer was a goal when design patterns were introduced in the first place (Gamma *et al.*, 1998), but large, printed design pattern catalogues have made this idea impractical.

While our approach offers promising answers and results, it also triggers new questions and further research to be done. One thing to be done is definitely to improve the formula for guiding the algorithm. Another detail that could be addressed even more is research on threshold values – how many answers would be enough to further improve the quality of the dialogues? The approach could also be applied to development tools as a handy plug-in while developers are in doubt about which, if any, design pattern to use. The approach could also be applied to other, non-design pattern-related fields. The experiment may also have to be repeated with a larger knowledge base. Although the formulas were designed not to be affected by knowledge base size, it would be interesting to see how dialogue length is affected by a larger question-answer corpus (i.e. more than 1000 independent hints). We designed the formulas and algorithms in such a way that a larger knowledge base provides more opportunities for lower entropy in order to propose the pattern. Thus, we expect that having more questions can only be a benefit.

In this paper, we have shown that the OBDPR is not only a platform for the easy exchange of knowledge and experiences about design patterns; its ontological data layer also enables some intelligent services. One of those is definitely the algorithm that takes several hints from several authors and connects them into a guided dialogue. During the time that we performed this research, and while writing this paper, we repeated the experiment several times (with students, as a part of software development courses). We observed the same results (with slight but insignificant deviations) as presented in this paper. Industry partner responses to our research have also been positive.

References

Birukuo, A., Blanzieri E., Giorgini P. (2006). *Choosing the Right Design Pattern: The Implicit Culture Approach*. Technical report DIT-06-007. Department of Information and Communication Technology, University of Trento.

Core J2EE Patterns. http://www.corej2eepatterns.com/. Last visited in July 2013.

Dietrich, J., Elgar, C. (2005). A formal description of design patterns using OWL. In: *Proceedings of the 16th Australian Software Engineering Conference*, IEEE Computer Society, pp. 243–250.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1998). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gomes, P., Pereira, F.C., Paiva, P., Seco, N., Carreiro, P., Ferreira, J., Bento, C. (2003). Selection and reuse of software design patterns using CBR and WordNet. In: *Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering, SEKE 2003*, pp. 289–296.

Ittycheriah, A., Franz, M., Zhu, W., Ratnaparkhi, A., Mammone, R.J. (2001). Question answering using maximum entropy components. In: *Second Meeting of the North American Chapter of the Association for Computational Linguistics on Language Technologies 2001, Pittsburgh, Pennsylvania, June 1–7, 2001*. North American Chapter of The Association for Computational Linguistics, Association for Computational Linguistics, Morristown, NJ, pp. 1–7. doi:10.3115/1073336.1073341.

Kampffmeyer, H., Zschaler, S. (2007). Finding the pattern you need: the design pattern intent ontology. In: *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Vol. 4735. Springer, pp. 211–225.

Kung, D.C., Bhambhani, H., Shah, R., Pancholi, G. (2006). An expert system for suggesting design patterns: a methodology and a prototype. In: Khoshgoftaar, T.M. (Ed.), *Software Engineering with Computational Intelligence*. Kluwer.

Pavlič, L., Heričko, M. (2008). Semantic web as an enabling technology for better design pattern adoption. In: *19th Central European Conference on Information and Intelligent Systems, Varaždin, Croatia*. University of Zagreb, Faculty of Organisation and Informatics.

Pavlič, L., Heričko, M., Podgorelec, V., Rozman, I. (2009). Improving design pattern adoption with an ontology-based repository. *Informatica*, 33(2), 189–197.

Pavlič, L., Podgorelec, V., Heričko, M. (2014). A question-based design pattern advisement approach. *Computer Science and Information Systems*, 645–664.

Shannon, C.E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27.

W3C Semantic Web . https://www.w3.org/standards/semanticweb. Last visited in March 2016.

L. Pavlič received his PhD degree in computer science in 2009 from the University of Maribor, Slovenia. He is currently a senior researcher with the Institute of Informatics, FERI, at the University of Maribor. His main research interests include all aspects of IS development, reuse in software engineering, object orientation, information system architecture, Java, UML and XML-related technologies, semantic technologies, intelligent systems, big data and nosql. He has appeared as an author and co-author in several peer-reviewed scientific journals. He has also presented his work at a number of international conferences. In addition, he has participated in many national and international research projects.

M. Heričko is a full-time professor at the Institute of Informatics. He is the head of the Information Systems Laboratory and head of the Institute of Informatics. He received his PhD in Computer Science from University of Maribor in 1998. His main research interests include all aspects of information systems development, software and service engineering, agile methods, process frameworks, software metrics, functional size measurement, SOA, component-based development, object-orientation, software reuse and software patterns. Dr. Heričko has been a project or work co-ordinator in several applied projects, project or work co-ordinator in several international research projects and committee member and chair of several international conferences.

V. Podgorelec is a professor of computer science at the University of Maribor, Slovenia. His main research interests include intelligent systems, semantic technologies, and medical informatics. He has been participating in many international research projects and is author of several journal papers on computational intelligence, software engineering and medical informatics. Dr. Podgorelec has worked as a visiting professor and researcher at several universities around the world, including University of Osaka, Japan, University of Nantes, France, University of La Laguna, Spain, University of Madeira, Portugal, and University of Applied Sciences Seinäjoko, Finland. He received several international awards and grants for his research activities.

P. Repolusk received her PhD degree in mathematics in 2013 from the University of Maribor, Slovenia. She is currently working as a teaching assistant at Faculty of Natural Sciences and Mathematics at the University of Maribor. Her main research interests include combinatorics, particularly graph theory, especially different types of domination and convexity. She has appeared as an author and co-author in several peer-reviewed scientific journals and presented her work at a number of international conferences.

Entropija grindžiamas algoritmas, skirtas tinkamiausiems tipiniams projektavimo sprendimams parinkti

Luka PAVLIČ, Marjan HERIČKO, Vili PODGORELEC, Polona REPOLUSK

Straipsnyje sprendžiamas uždavinys, kaip projektuojamai programai pasirinkti tinkamiausius tipinius projektavimo sprendimus. Žinomų tipinių programų projektavimo sprendimų skaičius sparčiai auga, bet esamos jų tvarkymo ir paieškos priemonės atsilieka nuo šios spartos. Straipsnyje pasiūlyta tinkamiausių tipinių projektavimo sprendimų paieškos ir žinių apie tokius sprendimus mainų platforma. Pasiūlytoje platformoje projektuotojui nereikia pačiam ieškoti jam tinkamiausio tipinio projektavimo sprendimo. Vietoje to dirbtinio intelekto metodus naudojantis platformos komponentas klausia projektuotojo, kokią projektavimo problemą jam reikia išspręsti, pats pasiūlo jam tinkamiausius problemai spręsti tipinius sprendimus. Kitaip nei kitos panašaus tipo priemonės, mūsų pasiūlytas komponentas nėra sukurtas kaip ekspertinė sistema. Tokia sistema veikia per lėtai. Mes naudojame kryptingą dialogą, kuriame klausimai generuojami automatiškai kombinuojant įvairius skirtingų autorių parengtus informacijos šaltinius. Straipsnyje išsamiai aprašyti tam naudojami algoritmai su atitinkamomis formulėmis. Straipsnyje taip pat yra aprašyti eksperimentai, vykdyti mūsų komponento ir sprendimų medžio metodą naudojančios ekspertinės sistemos veikimui palyginti.