An Expert System for Design Patterns Recognition

Omar AlSheikSalem[1] and Hazem Qattous[2]

[1] Department of Software Engineering, Applied Science University, Amman, Jordan
[2] Department of Computer Information Systems, Applied Science University, Amman, Jordan

Summary
Several design problems are faced during the designing process of systems. Many of these problems are faced several times throughout the work of designers and programmers during the design and development stage of different systems. Through their work, experts can notice similarities between different design problems contexts and reuse the same previous designed solutions each time. They can notice similarities between different problems contexts because they have the experience to do so. Unfortunately, novices in design cannot recognize these similarities to benefit from using previous solutions to solve their current problem(s).
Design patterns are one of the ways to encapsulate the experience of experts as they are considered to be design solutions for recurrent design problems. To use design patterns in solving design problems, a designers should have the experience to discover and recognize the applicability of a specific design pattern that is suitable to solve a specific design problem. This experience is not available to novices in the design and programming fields. Reading documentations of the design patterns from their catalogues is almost the only way that could be followed by novices to take the advantages of applying design patterns solutions to their design problems.
This paper investigates the involvement of a rule-based system to assist novices in exploring their design problems. The developed system operates by the dependence on asking the user some questions through which the system can notice the similarities between the current problem and one of the previously solved problems using design patterns. As a result, the developed system, by answering the questions, can recognize a specific suitable design pattern to solve a specific design problem. Throughout the paper, ten design patterns are selected as a representative sample to conduct the investigation.

Keywords:
Design patterns, Novice, Rule based system, Expert Systems.

1. Introduction

Experience gained by programmers and designers through their work allows them to identify similar problems in different situations. That would help them in applying the same standard solutions for the identical problems that they have solved in the past. This is reflected on their work in being more effective as they save the effort required for analyzing the problem again to be solved [7]. Therefore, the experience provides the experts with reusable, elegant and high quality design solutions
These solutions are not available to novice programmers and designers because they have not faced many problems that required solutions. Consequently, they do not have the ability to notice the similarities between various problems and provide the best solution(s) to their design problems. Such a problem may cause a delay in work, consume time, and characterize the work with a low efficiency and a poor quality. Part of the experts experience in the design problems and solutions field has been documented and encapsulated into a software design patterns. Software design patterns are design solutions that can be used repetitively to solve similar problems that are encountered during the design process. Their importance arises from the fact that they provide constant solutions for problems that different designers may face. Their importance is extended to include providing a very high quality reusable design solutions and provide these solutions to designers and programmers who are not aware of it [7].
Novices' problem of not having the required experience to solve their design problems using previously designed solutions could be solved by providing novices with the software design patterns. Providing experience through design patterns is something common and documented, as of the examples by [7] [6]. Grand [7] says that one of the benefits of "putting patterns into words" or describing them is giving the experience to novice designers and programmers who have not discovered the patterns yet.
It could be noticed that design patterns documentation is almost the only way that is common for design patterns presentation. Attaching an example to clarify the context into which a design pattern is used, is a common thing throughout the documentation of design patterns. Design patterns documentation in Gamma's et. al. and Grand's [6] [7] contains an example attached to each design pattern, which indicates the importance of these examples in clarifying the use of design patterns.
To recognize a design pattern applicability into a specific design problem, a novice should read a design pattern documentation. This leads to a waste of his/her time, effort and delays the work. In addition, reading design patterns and understanding them is not an easy task for novices who do not know about design patterns and have no experience in the background knowledge that is required to understand what has been documented. As an example, many literatures include the knowledge of object-oriented programming as the main base to understand and be able to apply design patterns. Astrachan et. al. [2] criticizes the way of presenting design patterns in literatures including the

Manuscript received January 5, 2017
Manuscript revised January 20, 2017

catalogues, as they are not easy to be understood by people who are unfamiliar with object-oriented programming. Stuurman and Florijn [12] also presented the same problem of design patterns.

It could be easily concluded that there is a problem facing novice programmers and designers in acquiring the required experience to adopt previously solved problems into their design solutions. This is the motivation to investigate and introduce a tool that supports design patterns recognition in problem contexts. It is also an encouragement to make it easier for novices to handle design patterns and adopt them as solutions for their design problems.

This paper aims to find a step towards helping, assisting and guiding novice designers and programmers in solving design problems that already have possible solutions through recognizing the applicability of a specific design pattern into a specific design problem. This paper introduces rule-based system (Expert System) as a solution for the problem. However, the question is how can rule based system help novices to identify and recognize a software design pattern applicable to a particular software design problem? From the literature, an Expert System is defined "as a computer system (hardware and software) that simulates human experts in a given area of specialization" [8].

This paper, as an introduction, investigates a sample of ten design patterns. The chosen design patterns are those which are used extensively as examples in teaching design patterns to novices, especially students, as they are considered the novice designers and programmers in different researches. The same way of choosing a design patterns sample was followed by Lewis et. al.[9].

From different literatures, mainly the GoF book [6], it could be concluded that the following design patterns are of the most used and useful for novice designers and programmers:

1. Creational Design Patterns
   - Factory Method.
   - Singleton.
2. Structural Design Patterns
   - Façade.
   - Decorator.
   - Composite.
3. Behavioral Design Patterns
   - Observer.
   - Visitor.
   - Iterator.
   - Command.
   - Strategy.

The above design patterns are adopted to be investigated for the purpose this paper. Each design pattern problem characteristics are discussed and formalized into a questions format. For each design pattern, several questions are suggested and discussed. They are tailored to be easily answered and understood by novices. Their creation also depends on their ability to express the design pattern from which they are extracted. Their ability to distinguish a design pattern from others is also taken into consideration.

The main contributions of this paper can be summarized as following:

1) Extracting the required questions to identify selected design patterns from each design pattern documentation.

2) Implementing an expert system that can help novice programmers and designers to solve their design problems by directing them to use the suitable design patterns where appropriate.

2. Design Patterns Under Investigation

2.1 Creational Design Patterns

Creational patterns provide procedures to create objects when their creation requires making decisions. These decisions will dynamically decide which class to be instantiated or to which object an object will delegate the responsibility. The value of creational patterns is that they provide a way to structure and encapsulate these decisions [7].

Suggested Questions:

- Do you need to create an object, which could be a part of a hierarchy class structure or you need to create an object and ensure that it is only created once?
- Do you need to create objects or instances and their creation needs decisions?

2.1.1 Factory Method Design Pattern

In this pattern, the decision of which class object is needed to be created is encapsulated in its own class. Therefore, neither the client nor the data source will be aware of the actual type of object created [7]. This pattern helps in creating an object, which at a creation time, can allow its subclasses decide which class to instantiate [11].

Suggested Questions:

- Do you need to create an object to represent external data or process an external event where another object is responsible for creating and determining the type of this object?
- Do you need to create different object types using an object responsible for determining the object type to be created?
- Do you need to instantiate an object in a class hierarchy where the number of subclasses of this hierarchy is not so big?

2.1.2 Singleton Design Pattern

The intent to use singleton design pattern is the need to provide a single instance of a class and ensure that this instance is only created once. Its intent could be extended to provide a global single point of entry or access to this sole instance or object. The use of it is common in the context where it is important to create exactly one instance from a class and this instance should be easily accessible by all other objects, use the singleton object [6][7].

Suggested Questions:

- Do you need to create an exactly one instance from a class and ensure that it is the only instance created from this class and no more instances will be created from the same class?
- Do you need to create an instance from a class and provide only one access point to it so all other instances or objects of other classes can access it through the same solely entry point?
- Do you need to create an instance from a class, ensure that no more instances will be created from the same class, and provide a single point of entry to it for all other instances?
- Do you need to create a single instance from a class and provide global point of access to it?

2.2 Structural Design Patterns

This type of patterns generally provides different ways to allow different objects to communicate with each other in an organized manner [7]. They also provide a way to compose objects and classes to form larger structures. Structural design patterns compose interfaces or implementations through using inheritance, an example of that is the multiple inheritance [6] [7].

Suggested Questions:

- Do you need to create an object that is composed from different objects?
- Do you need to compose different objects at run time?
- Do you need to compose a class from different classes through inheritance (multiple inheritance as an example)?
- Do you need to create a class that is considered as a composite of different other classes as it has all their properties (using multiple inheritance as an example)?
- Do you need to create an object that is composed from different other objects and this object has new functionalities over all the other of its components objects?
- Do you need to create an object that could be composed from different other objects at run time?

2.2.1 Composite Design Pattern

Composite pattern provides a solution for organizing a hierarchy of objects. This pattern is applicable in the case of the ability to organize objects in a tree-like manner. It adds the consistency advantage in manipulating objects by supplying a common interface or super class to all of them in the tree-like structure [7]. The intent of it is to represent a part-whole structure by composing an object into a tree structures. Composite pattern describes and solves two main problems in classes and objects relations. The first is the ability to compose an object from other objects, which are inherited from the same interface. The second is dealing and solving the recursive composition into a simpler manner [6].

Suggested Questions:

- Do you need to build complex objects, which may compose similar objects?
- Do you want to represent a tree like connected objects that consist of objects containing similar objects in a recursive manner?
- Do you need to create an object or objects that composed from other objects inherit the same interface that the object you want to create also inherits?
- Do you need to create an object(s) that is a part of a hierarchy structure and it is considered as a node in this structure that is composed of or has leaves which are also part of the upper node that the object need to be created is a chilled of?
- Do you need to create an object or objects that inherit an interface and composed from other objects that inherit the same interface?
- Do you need to create an object, which is part of a tree like structure and is composed from other objects in the same tree like structure?
- Do you need to create an object that has a recursive relationship as one of its subclasses is itself?
- Do you have the situation where some classes inherit a main super class and one of these classes has children which including the same other classes that inherit the main super class?

2.2.2 Façade Design Pattern

Façade pattern offers the design the advantage of hiding the complexity of related objects communications. It provides an interface that contains all the needed specifications to deal with the other objects. Other objects will use that

interface to communicate with a set of objects that are hidden by it [7]. In general, façade provides a single point of entry to a subsystem or shows the services that can be provided by a subsystem and determine its use through the provided functionalities [6].

Suggested Questions:

- Do you have a big system and you want to divide it into smaller, simpler and easier to be managed subsystems?
- Do you need to reduce the complexity and communication dependencies between different subsystems?
- Do you have a subsystem (could be a package) composed of set of classes or objects and you need to provide a single way to communicate with them through an interface?
- Do you have a subsystem that other objects or subsystems need to communicate with it?

2.2.3 Decorator Design Pattern

The intent of the decorator pattern is to provide an additional functionalities and responsibilities dynamically to an object. It provides a flexible alternative for sub-classing to extend the functionality of an object [6]. It extends the functionality of an object in a transparent way to its client so; it will not affect other objects [7].

Suggested Questions:

- Do you have too big hierarchy class structures and too many sub-classing because of too big alternatives in responsibilities and behaviors of an object?
- Do you need to create an object that needs different functionalities and responsibilities to be added to and withdrawn from it dynamically at run time?
- Do you have a situation that needs to let user to choose different functionalities to be added to or withdrawn from an object dynamically at run time?

2.3 Behavioral Design Patterns

This class of design patterns contains patterns that describe the communication between objects and the assignment of responsibilities between them in addition to that it is concerned of algorithms. It uses inheritance relationship between classes and object composition to distribute behavior between them [6].

Suggested Questions:

- Do you need to provide some algorithms into your program or do you need to distribute a specific behavior between classes and make it specific or do you need to perform a task through cooperation between different objects where one object cannot perform it?
- Do you have several objects need to be interacted which may lead to provide a tightly coupled design?

2.3.1 Iterator Design Pattern

The Iterator pattern provides a way for sequentially accessing a collection of objects by another object. The object that accesses the collection does not realize the structure or class of the collection and can only access the collection through an interface that this pattern provides. Its access will be performed independently from the class that implements the interface and the class of the collection [7]. Freeman et. al. [5] introduce this pattern as it provides the ability to access an array and manage that access by controlling the movement of the iterator itself.

Suggested Questions:

- Do you need to access an aggregation object contents without exposing the internal structures of that object?
- Do you need to access an aggregation object sequentially?
- Do you have an array or a list or any type of such aggregation structure that you need to access it sequentially?

2.3.2 Observer Design Pattern

This pattern provides the ability of objects to register with another single object. Registering means that when the state of that object change, the registered objects will be notified [7]. Observer pattern defines the one-to-many relationship between objects [6]. In general, such design pattern is used when there are several objects, which depends on each other. To eliminate their dependencies on each other, an object, which is the subject, is created. Each of the registered objects transfer its dependency to the subject object. One example of using observer pattern is thread programming.

Suggested Questions:

- Do you have a situation where some objects are depending on the states of other objects?
- Do you have objects that need to monitor or observe the changes in the state of each other and need to be notified whenever the state of any of them has been changed?

2.3.3 Visitor Design Pattern

The Visitor pattern provides a way of separating the logic of an operation implementation that involves a complex structure. If visitor pattern is not used then the logic is needed to be implemented in each class of that complex structure to support that operation [7] [6]. Optimal visitor design pattern is expressed as two class hierarchies. One is for the visitor and its subclasses and the other is for the element and its subclasses. The element hierarchy is the one

on which the methods or functions in the visitor hierarchy are applied. The main point that both, [7] [6], concentrates on, is the use of visitor design pattern whenever there is a complex structure. Gamma et. al. [6] defines the complex structure as either composite (meaning Composite design pattern) or a collection such as the array or the list.

Suggested Questions:

- Do you have a class hierarchy where there are functions, methods or behaviors that are shared between all of the subclasses; however, there is a need to implement these functions, methods or behaviors differently in each subclass?
- Do you have a class hierarchy and need to apply different functions, methods or behaviors on it where each function, method or behavior is applied on all concrete classes of the class hierarchy?
- Do you have a complex structure (a hierarchy class structure or an array or list) that contains different elements and need to apply the same different functions, methods, behaviors on these elements?

2.3.4 Strategy Design Pattern

The Strategy design pattern provides a way to encapsulate different algorithms [7] or a family of them [6]. Grand [7] replace the term "family" with the term "related algorithms". Both terms gives the impression that the algorithms this pattern deals with, in general performs the same task but in different manner. Strategy pattern could be used if there are many related classes and they are different only in their behavior or if there is an algorithm and has different variants. It also could be used to separate between the client that uses an algorithm in different variant and the algorithm itself. Therefore, the client is not concerned or knows about the structure of the algorithm or its specific data structures [6].

Suggested Questions:

- Do you have several related classes that are different in their behavior?
- Do you need to provide a choice of related classes' collection that almost have the same responsibilities but perform them in different manners?
- Do you have different algorithms that are performing almost the same task but in different behavior or manner?
- Do you need to hide the complexity of different algorithms from the client?
- Do you have a class hierarchy that represent different algorithms?
- Do you need to use a family of algorithms and make them interchangeable?
- Do you have the situation where there are many behaviors or algorithms that are separated by conditional statements and can be separated into different classes?
- Do you have a family of different related classes or algorithms that perform almost the same task into different manners and you need to use them interchangeably?

2.3.5 Command Design Pattern

The Command pattern is used when there is a need for storing a command and there is a need to use this command several times later on. The pattern also is used when there is a need to perform some other operations on a command such as queuing, undoing or redoing it. Command pattern provides a way to invoke all transactions in the same manner because of its common Command interface [6] [7].

Suggested Questions:

- Do you have the situation where there is an action(s) needed to be issued or performed several times, so it is needed to be stored and recalled when needed later on?
- Do you have the situation where there is an action(s) and you want to provide the ability to queue (and/or) undo (and/or) redo this action(s)?
- Do you have the situation where you need to design, support or program transactions?

According to the above analysis and suggested questions that encapsulates the knowledge of applying design patterns, this paper investigates the ability to build a system that facilitates the use of design patterns by novices to solve their design problems that already have been solved by experts. The proposed system is considered as an expert system because it contains the required knowledge to apply design patterns where appropriate. The system is discussed in the following section.

3. The Expert System

To assist novice programmers and designers in applying a suitable design patterns where appropriate, an expert system prototype has been developed. The extracted and suggested questions presented above were inserted into the expert system to be able to guide the user. How the system works, simply, depends on asking questions to the user and then he/she replies. Each time the system asks a question and gets a reply, the system should be able to infer something, which is part of the solution. Therefore, with each answer, the system is gaining more knowledge about the problem and consequently, about its solution. The process can be described as, the system with the help of the user, searches the space of design patterns to find the suitable one to be

applied for solving the current design problem. The implemented expert system has been investigated against its suitability to be used as a guide to help novices in solving their design problem(s) using design patterns in a simple manner.

The structure of the expert system is composed of three main modules, a knowledge base, an inference engine and a user interface. Since this architecture is recommended by literature as Cawsey and Darlington [3] [4], it was adopted to be followed in developing the required expert system. The architecture is presented in the following figure (Fig. 1) as a package diagram. The prototype was developed using Java language. The architecture of the system was divided into three main components, Knowledge Base, Inference Engine, and User Interface. Components architecture facilitated the extendibility and traceability of the program. The knowledge base of the system contains all of the questions emerged from the above analysis. One of the challenges faced implementing the system was to represent knowledge in a knowledge base. A Rule and Fact classes were used for this purpose. Rules were represented as strings into the Rule class where objects of it were hold into the knowledge base. That was applied also to Facts; Facts were also used in this prototype to represent and express the questions.

The prototype works as follows; first, the rules were transferred from the knowledge base to the inference engine. This process came up with good results and eliminated the concern regarding inefficiency of evaluating IF statements. This process also converts the concrete rules format from writing each rule as standalone rule to a format of procedural connected rules.

Connecting rules in this format allows better efficiency in searching the goal state. This is because it takes the advantage of concentrating the search for the required design pattern within the design patterns under the same class or classification. As an example, the search for a Factory Method design pattern will be only within the class of Creational design patterns. Consequently, the search will be concentrated within the rules that lead to conclude one of the Creational design patterns instead of searching all of the rules of the system. This process reduces the search space for the required design pattern.

Another challenge faced during implementing the expert system is that the questions are too long, as text, to be represented into the prototype. It was noticed that introducing the actual questions into the rules leads to complexity and reduces the ability of tracing. To solve this problem, logical rules with logical names for questions were introduced. As an example, the logical name of the question that asks about Creational design patterns is ‘creational question’. Adopting the questions logical names allows introducing them in the logical structure of IF/THEN/ELSE in the inference engine and leaves the concrete questions in the knowledge base. Knowledge base kept the facts as strings into a HashMap data structure, which has two columns, key and value. Facts represents the questions that will be asked to the user and they are considered as the value in the HashMap. The keys are the logical names of the questions.

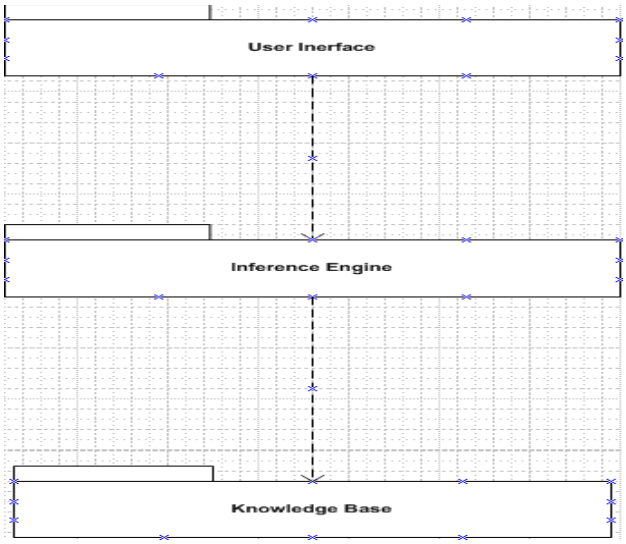

Fig. 1 Rule Based System Architecture

The implemented system also allows the user to ask 'Why?' and 'How?' questions. The 'Why?' question allows the user to query about the purpose of the current question introduced by the system. Which means if the system introduces a question to the user to be able to recognize a design pattern, the user can ask the system 'Why do you ask this question?'. In return, the system provides explanation for introducing such a question. Of course, the 'Why?' feature is only available after the user answers the first question to allow the system infers something before being able to provide justification for introducing the second question. The 'How?' question allows the user to query about 'how did the system reach to this conclusion?'. Apparently, the user can ask this question only when the system reached the conclusion, which is the recognized design pattern. The 'Why?' and 'How?' questions provide the transparency of the system work, questions, and its inference steps. It should be noticed that the system does not always provide a conclusion. This can happen for several reasons. One is that the user does not provide correct answers on the system questions. Another reason is that the required design pattern does not exist in the knowledge base of the system. A third reason is that the design problem does not need a design pattern application. In such cases where the system is not able to infer, it shows an apology message to the user.

The system components work can be summarized as the following. Knowledge Base component in the system holds all the questions that should be asked to the user to

recognize the required design pattern. It also shows all the design patterns that could be recognized by the system through holding logical names of the design patterns. Inference Engine component holds all the rules that are applied to reach the required design pattern. It uses the questions from the knowledge base and the answers on them from the user through GUI to infer a specific design pattern that solve a specific design problem. GUI component takes the responsibility of providing the user with interface through which he/she can see questions, answer them, and ask the system 'Why?' and 'How?' questions. It also shows the user different inferences done by the inference engine, the final result or conclusion which is the recognized design pattern and explanations of the user questions to the system ('Why?' and 'How?' questions).

The following is a demonstration of an example that shows part of the functionalities of the developed expert system.

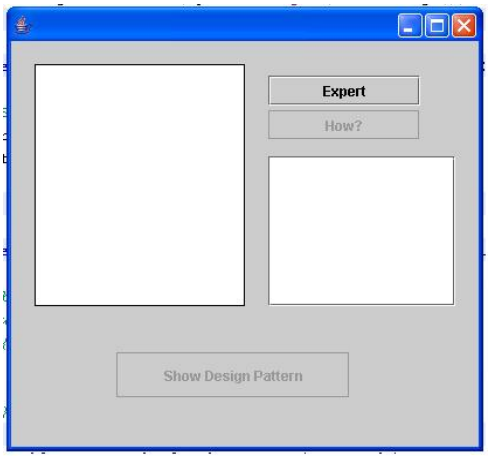

Fig. 1 Step 1, the user run the program and step2, the user press the Expert button.

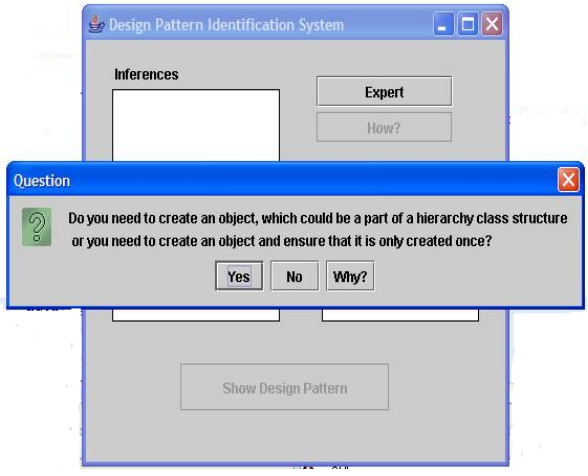

Fig. 2 Screen after the user has pressed Expert button and the first question is shown to him/her. Step 3, the user answer the first question by pressing 'Yes' button.

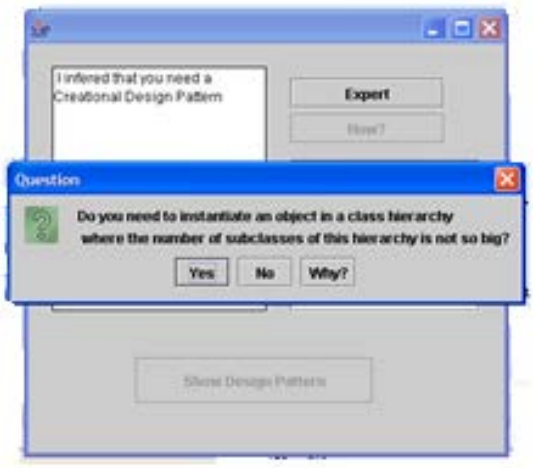

Fig. 3 The screen after answering the first question by 'yes'. The inference is shown in the back screen and the second question is shown in the fore screen. Step 4, the user answers the second question by pressing the button 'Yes'.

Fig. 4 The screen after answering the second question as 'yes'. The results are shown; the system inferred the need for Factory Method Design Pattern and shows a message telling that to the user.

4. Evaluation

To evaluate the prototype, a task of designing a small mobile controlling application was formulated. The requirements of the mobile application were documented. The documentation took into consideration including the applicability of three design patterns from those of which the prototype can recognize. The three design patterns are the Factory method, the Iterator and the Decorator. Four novice software engineering students at a University were

exercised as subjects. The subjects were selected from those who have finished the software design course in which design patterns are discussed briefly. The course is a third year level. The subjects were asked to make a class diagram design for the mobile controlling application. They were also instructed to use the expert system to improve their design by applying design patterns. The subjects were trained to use the prototype to be able to deal with it during the task. A time limit of two hours were given to the students and the four students were conducted the experiment in one session. They were not allowed to talk to each other during the session.

Analyzing the design generated from the students showed that only one student could apply the three required design patterns. The other three students could apply two of the three required design patterns. During an interview with each subject alone after the task, all the students were positive regarding the help they gained from the prototype. They believe that the expert system directed them to apply design patterns where appropriate. However, they also commented that some questions were not easy to be understood. They also mentioned that the prototype in many cases could not infer anything.

5. Conclusion

Throughout this paper, an investigation was conducted to study the ability of providing novice designers and programmers with the required experience to perform and solve their design problems that already have been solved by others. This is done by introducing the design patterns to them in a simpler manner than the design patterns catalogues do. The investigation took the following steps to be conducted:

Ten design patterns were investigated as a sample and one or more questions were evolved for each of them. The questions of each design pattern asks about the design problem context into which the design pattern is applied or created to solve. These questions are selected and formed to be simple, as they will be directed to novices, and can express the problems that design pattern can solve. So, the questions of each design pattern should be able, if answered correctly, to recognize the need to apply the design pattern into the design.

To proof the concept, an expert system prototype was implemented to be able to recognize ten of the most common design patterns. The system has three components, the knowledge base, inference engine and the GUI. Java was used as a programming language for system implementation. In general, the system works by introducing questions to the user. From the user replies, the system can search the space for the appropriate design pattern that can be applied. The developed expert system has a transparency feature as it allows the user to ask the system 'Why do you ask the current question?' and also allows the user to ask 'How did you reach your conclusion?'. The system, when used, either provides the user with a design pattern to solve the design problem or shows an apology message for not being able to recognize the pattern. The inability to recognize all of the documented design patterns and it's in ability to recognize design patterns correctly if the user does not answer all of the questions correctly, are the main weakness of the system.

One of the most important conclusions that were noticed throughout the investigation of this paper is the difficulty of evolving some questions that satisfy the requirements of keeping it simple and express the design pattern at the same time. It is concluded that such questions evolving need long experience in both, dealing with design patterns and dealing with their catalogues. Actually, this part of the research was the most difficult and time consuming.

The developed system was evaluated through a small empirical study. The study examines the use of the system in solving a real design problem by providing the subjects with a task of developing a class diagram design for a simple mobile controlling application. Results and interviews with subjects shows that the implemented expert system helped novices and directed them for applying the suitable design patterns.

Acknowledgement

The authors are grateful to the Applied Science Private University, Amman, Jordan, for the full financial support granted to this research.

References

[1] Sommerville, I., 2015. Software Engineering, 10th edition. Pearson Education.

[2] Astrachan, O., Berry, G., Cox, L., Mitchener, G., 1998. Design Patterns: an essential component of CS curricula [online]. Technical Symposium on Computer Science Education, Proceedings of the twenty-ninth SIGCSE technical symposium on Computer Science education, pp 153-160. Available from http://doi.acm.org/10.1145/273133.273182. [Accessed 14 January 2017]

[3] Cawsey A., 1998. The essence of artificial intelligence. UK: Prentice Hall.

[4] Darlington Keith, 2000. The essence of expert systems. England: Prentice Hall.

[5] Freeman, E., Freeman, E., Robson, E., Bates, B., and Sierra, K., 2004. Head First Design Patterns. O'Reilly Media, Inc.

[6] Gamma, Erich, Helm, Richard, Johnson, Ralph and Vlissides, John, 1994. Design patterns elements of reusable object-oriented software. UK: Addison Wesley.

[7] Grand, Mark, 2002. Patterns in Java, volume 1. USA: Wiley Publishing Inc.

[8] Castillo, E., Gutierrez, J., and Hadi, A., 2012. Expert Systems and Probabilistic Network Models. Springer Science & Business Media.

[9] Lewis, T, Rosson, M., and Quinones, M., 2004. What do the experts say? Teaching introductory design from an expert's perspective [online]. Technical symposium on computer science education, Proceedings of the 35th SIGCSE technical symposium on computer science education, pp 296-300. Available from http://doi.acm.org/10.1145/971300.971405. [Accessed 14 January 2017].
[10] Pressman, R., 2010. Software engineering: a practitioner's approach, 7th edition. London: McGraw-Hill Higher Education.
[11] Raj, Gopalan, 1999. The Factory Method (Creational) Design Pattern [online] available from: http://gsraj.tripod.com/design/creational/factory/factory.html [Accessed 14 January 2017].
[12] Stuurman, S., Florijn, G., 2004. Experiences with teaching design patterns [online]. Annual Joint Conference Integrating Technology into Computer Science Education, Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education, pp 151 – 155. Available from http://doi.acm.org/10.1145/1007996.1008037. [Accessed 14 January 2017]

Omar AlSheikSalem is currently an Assistant Professor at Applied Science Private University in Amman, Jordan. Dr. AlSheikSalem holds a Ph.D. degree in Computing from Bradford University, UK. His research interests are in Mobile TV, especially in the context of consumer needs and mobile advertisements, E-commerce, multimedia, Mobile TV content, and new ways of advertising and interactive video.

Hazem Qattous is currently an assistant professor at Applied Science University in Amman, Jordan. Dr. Qattous holds a Ph.D. degree in Computing from Glasgow University, UK. His research interests are in Human-Computer Interaction.