Just-in-Time-Teaching Experience in a Software Design Pattern Course

Ye Tao, Guozhu Liu
School of Information Science & Technology
Qingdao University of Science & Technology
266061, Qingdao, China
ye.tao@qust.edu.cn

Jürgen Mottok, Rudi Hackenberg
Laboratory for Safe and Secure Systems, LaS³
Ostbayerische Technische Hochschule (OTH) Regensburg
D-93025 Regensburg
juergen.mottok@hs-regensburg.de
rudi.hackenberg@hs-regensburg.de

Georg Hagel
Kempten University of Applied Sciences
Bahnhofstr. 61
87435 Kempten, Germany
georg.hagel@fh-kempten.de

***Abstract*— Teaching design patterns in computer science introductory sequence is more difficult than learning it. We present a Just-in-Time-Teaching (JiTT) practice in *Software Design Patterns* course targeted to entry level students. We proposed a model that combines JiTT concepts with didactic considerations, preparation materials, in-class sessions and post-lecture assignments. We have effectively used this approach to teach two undergraduate classes. The paper also presents the results of a survey conducted to identify the effectiveness of the JiTT activities through warm-up questions, class-wide discussions, experiments and exercises.**

Keywords—Just-in-Iime Teaching; JiTT; Activating Learning; Practical Learning; Design Pattern; Software Engineering

I. Introduction

Design patterns [1] have tremendously impacted the software development over the past decades. Besides preliminary and intermediate programmers, high-level talents are also required to complete the software architecture design and implementation.

As a follower research of the cooperation project between Qingdao and Regensburg Universities [2], on both sides, *design patterns* will be taught over three weeks. This paper presents the situation of applying JiTT pedagogical strategy in real teaching in Qingdao University of Science & Technology (QUST). We describe the details of the teaching activities and procedures, didactic considerations, learning objectives, reading material, measurement criterions and questionnaires. We also discuss the learning performance based on the outcomes and feedbacks.

Recent years, software engineering and design patterns are listed as main courses in most Chinese colleges. As a new course, its text books, overall arrangements and teaching/learning methodologies need be further explored and discussed. However, most higher education computer science courses are designed to teach programming languages and development techniques, rather than the fundamental ideas and basic principles of software engineering. Thus, learners' potentials are restricted, and the training result does not meet the actual needs in software industry [3].

II. Overview of the Course

As a mandatory course, *Design Patterns* closely links with other professional software engineering courses, and is usually taken by all undergraduates in CS. All the attendees have already taken appropriate prerequisite courses, and have learned at least one object-oriented programming language. The objective of the course is to teach attendees the correct way to create an object-oriented design. Attendees will understand what good design is as well as how to achieve it. We identified three levels of learning outcomes according to different target audience:

1) have some basic knowledge and understanding of the concept of software architecture and design patterns;

2) be familiar with several widely used patterns and be able to develop simple applications to illustrate learned patterns;

3) have the ability to consciously apply proper patterns in future software design process.

There have been attempts to teach *design patterns* within game design [4], in an independent course targeted to a small number of learners in intermediate level, which is quite different from the situation in our experiment. In the experiment scenarios of this paper, *design pattern* is taught as a chapter in the course of *software engineering*, which is also targeted at entry-level students. Due to the tight teaching schedule and its steep learning curve, it often makes freshmen very difficult to even partially grasp the essence of the design pattern.

As a design course, the prerequisite courses were experience with Object-Oriented Programming (OOP) and UML foundation. As these parts have already been described in previous courses, we start introducing patterns after recalling OOP fundamentals and principles. Table I gives an overview of the settings of the course, including the sections of the lecture and the tasks the students focused on.

TABLE I. Statistics of Warm-up Questions

Seq.	Section	Hours	JiTT[a]

1	OOP Fundamenetals		4	N
2	Design Principles		4	N
3	Strategy Pattern	Lecture	2	Y
4		Experiment	2	
5	Observer Pattern	Lecture	2	Y
6		Experiment	2	

[a] *JiTT pedagogy has been applied in this section*

A. *OOP Fundamentals*

Personal experience indicates that it could be better if the instructor leads a review of the OOP fundamentals, before introduce the patterns. The knowledge of an object-oriented language (Java/C#/C++) is important in order to understand the various implementation options that will be investigated later. In particular, participants must be familiar with inheritance, polymorphism, encapsulation, abstract classes and composition [5]. Because OOP mechanisms themselves do not portray the core spirit of the OOP design patterns, it is suggested that the instructor link these concepts in conceptual level, specification level and implementation level, to guide the students have a full understanding of the meaning of OOP.

B. *Design Principles*

Basic knowledge of OOP language (the concept of encapsulation, inheritance, polymorphism and etc.) only forms the foundation of learning design pattern, while a deep understanding of the OOP principle is the critical prerequisite to master the essence of the design pattern. In order to design a high-quality software module with flexibility, scalability, maintainability and reusability, it is important to have an in-depth understanding of the basic design principles.

In the introduction of principle, we suggest emphasize two basic principles [6] at first, *i.e.* "*program to an interface, not an implementation*", and "*favor object composition over class inheritance*", as they are essentially the basis of all other principles, and are widely used in dozens of design patterns. The rest principles can be explained from easy to difficult, following the order of SOLID [7] (*Single responsibility*, *Open-closed*, *Liskov substitution*, *Interface segregation* and *Dependency inversion*).

C. *Design Patterns*

It is necessary to clarify the relationship between designs and patterns. Each specific pattern describes a particular scenario with constraints, motivations and relationships, and provides a relatively optimum solution to these problems. From this point of view, we encourage students to apply patterns in practical problem analysis, rather than just understand or remember them. Therefore, we use case studies as a means for showing the definition, benefits, and costs of design patterns. When introducing a specific pattern, instead of directly presenting the final solution, we recommend the instructor explain a progressive evolution of entire design refactoring, in order to help students understand that design patterns are generalized from large amount of commonly occurring problems.

III. Teaching Activities

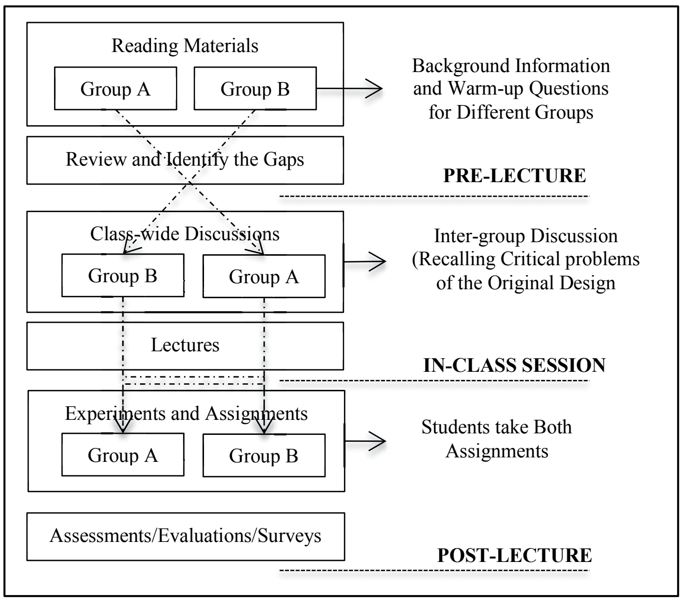
Just-in-Time Teaching founds in a constructivist [8, 9] based view point of students learning. The learners are engaged in a real-life context of problem solving. In our practice, the teaching processes are mixed with learning materials, warm-up questions, class-wide discussions, experiments/assignments and evaluations, as shown in Fig.1.

A. *Pre-lecture and warm-up questions*

We combine the descriptive paragraphs and warm-up questions in the form of "cloze", and students are expected to go through all the materials and answer the questions at the same time. They are given the framework and part of the code for a starting design, and asked to complete several critical elements that achieve the desired realization. As all the outside classroom activities are assigned through an e-Learning platform (*moodle* in our experiment), results and feedbacks are given immediately by the automated evaluation module. In this way, student will have the feeling that the teacher was adapting instruction in response to their learning needs.

1) *Strategy Pattern*

Some of our students have shown interest in playing, designing and developing mobile applications and computer games, including both business and technical aspects. *Angry Birds* is known by most attendees of the course. We use this game as an example to introduce the strategy pattern. We present some possible approaches in the reading materials to guide students to think about that each type of the birds has its unique characteristics, and in this case, how to design and implement behaviors of the birds.



Reading Materials
Group A
Group B
Background Information and Warm-up Questions for Different Groups
Review and Identify the Gaps
PRE-LECTURE
Class-wide Discussions
Group B
Group A
Inter-group Discussion (Recalling Critical problems of the Original Design
Lectures
IN-CLASS SESSION
Experiments and Assignments
Group A
Group B
Students take Both Assignments
Assessments/Evaluations/Surveys
POST-LECTURE

Fig. 1. Design pattern teaching models with JiTT.

Inheritance is the easiest way that most students will try at the very beginning. First, we show students a possible solution, *i.e.* creating one *Bird* superclass from which all other bird types inherit. Next, we prompt that the popularity of the game leads to different versions of extension and frequent updates, to make new types of birds become available. And as players advance through the game, more special abilities should be added or activated in various birds. To meet these requirements, students are asked to modify the starting design by filling some blanks in the code, helping them to understand that the use of *inheritance* has not turned out so well when it comes to maintenance. Finally, let the attendees consider and summaries the disadvantages of using *inheritance* to provide birds behavior, by answering some question from the text book [10].

2) Observer Pattern

Twitter, *WeChat* and other Social Network Service (SNS) become indispensable tools for students' daily communication. Most students are very familiar with the functionalities of *Subscribe*. Compared to most commonly used instances, *e.g.* Java Message Service (JMS) and Model-View-Controller (MVC), we found SNS is an easy-understand example to introduce the *Observer* pattern.

Similarly, we present the first implementation for a simple topic and observers can register to this topic. To avoid missing out when something interesting happens, whenever any new message will be posted to the topic, all the registers observers should be notified. Initially, we define three subscribers and let the students *manually* modify the code (usually copy and paste three lines). We then point out that with the increasing the number of the subscribers, the original design must be expandable, to provide a way to add/remove subscribers at run time, instead of the statically coding.

In the warm-up questions, students are required to make *some* (not all) of the birds explosible. When adding new behavior to the *Bird* superclass, they were also adding behavior that was not appropriate for some *Bird* subclasses. Thus, students have to override the *explode()* method in every subclasses.

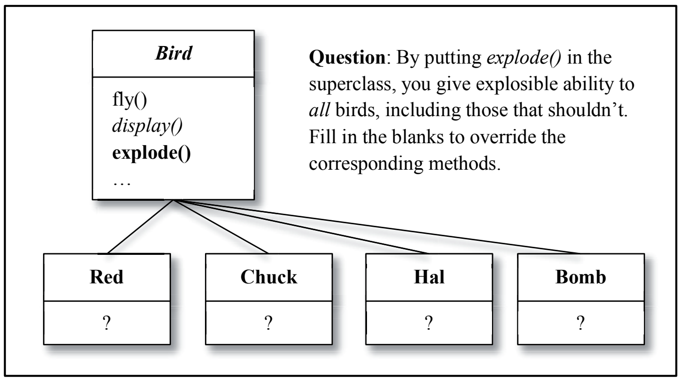

Bird
fly()
display()
explode()
…
Question: By putting explode() in the superclass, you give explosible ability to all birds, including those that shouldn't. Fill in the blanks to override the corresponding methods.
Red
?
Chuck
?
Hal
?
Bomb
?

Fig. 2. Design pattern teaching models with JiTT.

The statistics of the warm-up questions are listed in TABLE II. Note that all warm up questions in the pre-lecture activities are restricted to problem definition and its *first design* (without any patterns). Facility index (the mean score of students) indicates that up to 90% of the students understand the problem well, and are capable to modify the original design correctly to meet the requirements. Based on initial implementations, we encourage attendees to discover restrictions/disadvantages by themselves, rather than giving the final solution directly.

TABLE II. STATISTICS OF WARM-UP QUESTIONS

Average Grade of Attempts[b]	First attempts	89.23%
	Last attempts	90.28%
Facility Index	Strategy Pattern	89.73%
	Observer Pattern	91.02%

b. *total attempts: 146*

B. *In-class session*

Previous studies [11] show that solely traditional lecturing session does not stimulate in-depth thinking. It is particularly prominent in software engineering courses since students remain passive recipients to the majority of information presented over 1.5 hours. By throwing warm-up questions and mixing discussions, we are able to improve the effectiveness of the lectures. However, in our experiment, in-class activities are restricted by some constraints.

1) According to teaching plans, 2 patterns are instructed within a lecture without using any classroom communications systems.

2) The number of the participants in our experiment exceeds 100, which led difficulties for the teacher to manage peer instructions (PI).

Due to these factors, we executed a class-wide discussion instead of a standard PI. We randomly separate attendees into 2 groups (this was done automatically in pre-lecture stage by *moodle*), and allocated a particular design pattern (either *strategy* or *observer*) to just one group of users. Therefore, each student has already got some basic ideas of at least one pattern. After recalling the background of cases, we ask 2-3 students from each group to describe the restrictions of the original design they have taken, and try to give a refactoring approach. Class-wide discussion provides opportunities to think about each pattern in more detail. Additionally, as there might be more than one way to solve the problem, it helps the teacher to orient the class to explore and analyze alternative viewpoints, and finally choose the most suitable pattern by comparing the aspects of implementation, limitation and its final effects. Since students have experienced the full lifecycle of a design (from an earlier solution to refactored solutions), they are able to see the value of a pattern.

More than 85% students believed that the discussion time was sufficient. This partially benefits from pre-lecture tutorial materials and warm-up questions, which helps instructors saving the lecture time of repeating the background information of the study cases. Up to 90% attendees confirmed the open questions in pre-lecture stage were solved by in-class activities.

TABLE III. QUESTIONNAIRES ON IN-CLASS ACTIVITIES

Q1	*There was sufficient time for the topic "design patterns".*				
Score[c]	**5**	**4**	**3**	**2**	**1**
	77(57.9%)	39(29.3%)	14(10.5%)	3(2.3%)	0
Q2	*The lecturer was responsive to (remaining) questions.*				
Score[c]	**5**	**4**	**3**	**2**	**1**

	82(62%)	43(32.3%)	8(6%)	0	0

[c] *5-Strongly agree, 4-Agree, 3-Neither agree nor disagree, 2-Disagree, 1-Strongly disagree*

C. Post-lecture activities

A 2-hour problem based experimental section was assigned after the lecture. We immerse students in a real problem, and ask them to master each specific design pattern through hands-on practices, including outlining their solutions, figuring out the UML diagram and writing/testing the implementation.

The scale and difficulty of the case selection should be moderate. Cases in [10] are appropriate for entry-level learners, as most students should be able to solve the problems by applying the patterns within a given time. Examples of real problems that we have used include the *duck pond simulation game* (*Strategy* pattern), *weather monitoring application* (*Observer* pattern) and etc.

To help to instill the principles, students are given the skeletons of the design and considerable portions of the code. What they are asked to do is implementing the primary classes and critical methods of the design pattern. Typically, in the case of *Strategy* pattern, to complete the duck pond simulation game, attendees are asked to:

1) Implement the "flyBehavior()"/"quackBehavior()" interface of each particular behavior class which simply delegates the its flying/quacking behavior to the current Duck instance.

2) Modify the sub-classes of each type of Duck to use the abstract behavior delegator rather than the hard-coded.

3) Assign different behavoirs to concrete classes at run-time to swich strategies dynamically on the fly.

By taking series of experiments, students have learned the fundamentals of design patterns within the context of some classic applications. To reflect the practical emphasis, final assessment was split to 70% exam and 30% experiment, to satisfy University regulations. As shown in Fig.3, we have found that by completing the experiments, most students well understood the difference between the two patterns, and were able to apply these and other design patterns in subsequent courses.

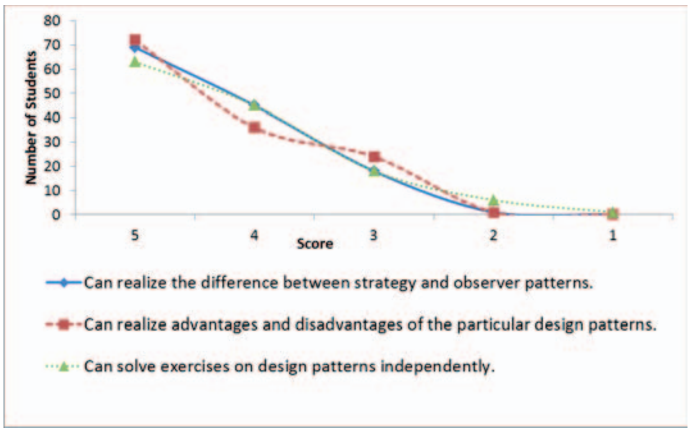


Fig. 3. Outcome evaluation after post-lecture activities.

IV. Evaluation and Discussion

The evaluation questionnaire includes a total of 31 questions divided into three groups, *i.e. Process and Structure*, *Comprehension of the Topic* and *Motivation*. The first group investigates the stages of JiTT activities as well as its basic concepts. The second group evaluates the teaching and learning outcomes. The last group is the satisfaction survey on the overall effects of the JiTT application experiments. 110 third-year students and 124 second-year students studied the course, in which 133 of them have participated the evaluation. We believe that all students answered the questionnaires honestly as they were aware that the score of the survey would not be included in their final assessment.

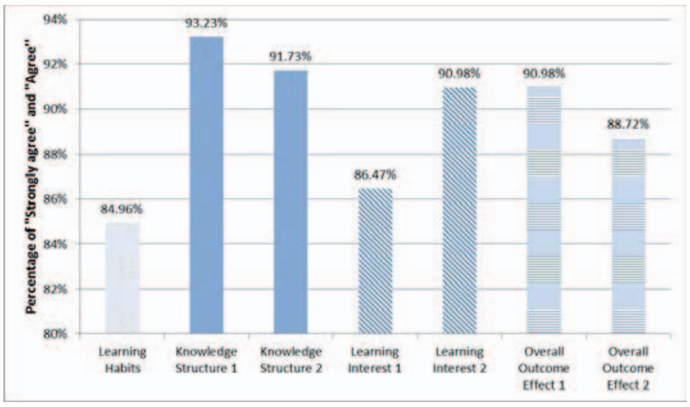


Fig. 4. Question matrix of the survey on JiTT in Software Design Pattern (*bars' shadings indicate different aspects of the survey*)

Fig.4 shows part of the evaluation results on JiTT effects from the aspects of learning habits, knowledge structure, learning interests and performance. Obviously, over 85% students reported that JiTT methods helped them improve their understanding of difficult concepts of patterns when compared with conventional lecture classes. Nearly 95% students reported that through pre-lecture questions, communicating sessions and post-lecture experiments, they learned something new and were able to tie design patterns with their previous knowledge. Up to 90% believed that the application of JiTT design patterns were fun, and sparked their interest in dealing with the topic further.

Besides of the above findings, we also proved that the concept of JiTT is comprehensible for most attendees, and there was a more/intensive communication between students and the lecturer than in other lectures without JiTT. Both the lecturers and the audiences had the feeling that the lecture-time was used more effectively with the insights of the preparatory stages as a foundation.

V. Conclusion

This paper presents the outcomes of our latest work in JiTT application in a software design pattern course. The critical procedures included mixing up reading materials and warm-up questions via online teaching platform, class-wide discussions and lecture time for in-class session, experiments and assignments after class, and a survey questionnaire. The workflow we have piloted works fine for entry level students in a relatively large class. Assessment has been conducted for over 130 students involved in this experiment. Statistics results demonstrated that JiTT concepts and its supporting activities have important effects on teaching and learning design patterns.

ACKNOWLEDGMENT

This paper is a following research of Chinese-German empirical case study starting from 2013. The research is supported by QUST Teaching Reform Program (No. 01051242), National Engineering Graduate Degree Teaching Committee Program (No. 2014-JY-058) and Shandong Province Graduate Education Innovation Projects (No.SDYC11035). Results are based on a series of experiments conducted in the autumn semester in 2014. The authors would like to thank all the students who participated in our studies on learning software engineering.

REFERENCES

[1] P. Wolfgang, Design patterns for object-oriented software development: Reading, Mass.: Addison-Wesley, 1994.

[2] Y. Tao, G. Liu, J. Mottok, R. Hackenberg, and G. Hagel, "Just-in-Time Teaching in software engineering: A Chinese-German empirical case study," in Global Engineering Education Conference (EDUCON), 2014 IEEE, 2014, pp. 983-986.

[3] S. Stuurman and G. Florijn, "Experiences with teaching design patterns," in ACM SIGCSE Bulletin, 2004, pp. 151-155.

[4] S. Bjork and J. Holopainen, "Patterns in game design (game development series)," 2004.

[5] N. Pillay, "Teaching Design Patterns," in Proceedings of the SACLA conference, 2010.

[6] B. I. Witt, F. T. Baker, and E. W. Merritt, Software architecture and design: principles, models, and methods: John Wiley & Sons, Inc., 1993.

[7] G. Booch, Object Oriented Analysis & Design with Application: Pearson Education India, 2006.

[8] G. Hagel, J. Mottok, M. Utesch, D. Landes, and R. Studt, "Software engineering lernen für die berufliche Praxis-Erfahrungen mit dem konstruktivistischen Methodenbaukasten," in im Tagungsband des Embedded Software Engineering Kongress, 2010.

[9] G. Hagel, J. Mottok, and M. Müller-Amthor, "Drei Feedback-Zyklen in der Software Engineering-Ausbildung durch erweitertes Just-in-Time-Teaching," in SEUH, 2013, pp. 17-26.

[10]E. Freeman, E. Robson, B. Bates, and K. Sierra, Head first design patterns: " O'Reilly Media, Inc.", 2004.

[11]J. Biggs and C. Tang, Teaching for quality learning at university: McGraw-Hill International, 2011.