A Pattern-based Model Transformation Approach to Enhance Design Quality

Yong-Yi Fanjiang[1] Nien-Lin Hsueh[2] Jonathan Lee[3]

[1]Department of Computer and Information Science, National Taichung University
[2]Department of Information Engineering and Computer Science, Feng Chia University
[3] Software Engineering Institute, National Central University

Abstract

Recently, the impact of design patterns on software quality has attracted a gradual attention since design patterns encapsulate valuable knowledge to resolve design problems and improve design quality. As an attempt towards the investigation of applying goals and design patterns to realize the model transformation, we proposed, in this paper, a goal-driven model transformation by applying design patterns to transform an analysis model into its corresponding design model with an emphasis on the non-functional requirements. The use of goals makes it easier to transform the functional and non-functional requirements into the software models, and derives the candidate design patterns to help satisfy non-functional requirements for resolving the design problems and improving software quality.

Keywords: Design patterns, model transformation, design quality, nonfunctional requirements.

1. Introduction

The software quality has long been recognized as an important topic since the early days of software engineering. In the past, researchers and practitioners alike have examined how systems can meet specific software quality requirements. Therefore, a growing number of practitioners have shown great interests in using design patterns towards high-quality software, since design patterns represent high-level abstractions that reflect the experiences of no other than skilled practitioners themselves. Design patterns have become a popular means to encapsulate object-oriented design knowledge. They capture successful solutions to recurring problems that arise when building software systems.

In this paper, we adopt the goal-driven use case (GDUC) [14] and fuzzy object-oriented modeling (FOOM) [15] served as the formal specifications to capture and specify imprecise requirements, and provide a set of pattern-based transformation rules to

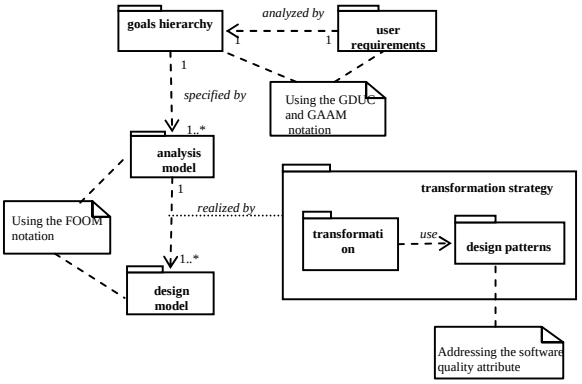

Figure 1: An overview of our proposed approach

deal with the software quality issues (see Fig. 1 for an overview). In Fig. 1, the designer constructs the goals hierarchy based on goals analysis and goals interaction. Before applying the design pattern transformation rules, the designer selects a suitable design pattern from the candidate design patterns through matching the non-functional goals and intents of design patterns, and then transforms the analysis model into design model by applying the transformation rules incrementally. Our proposed approach has the following features:

- Goal-driven use cases and goals hierarchy are built according to the goals and use cases analyzing. The analysis model is constructed based on the goals hierarchy by using the FOOM notations and semantics to specify various alternative models.
- Design patterns serve as a supporting design knowledge base to help the development of a system. A goal-driven method is provided to facilitating the selection of candidate design patterns from the analysis model, and the design pattern transformation rules are used to helping the enhancement of non-functional requirements and improve the software quality.

The organization of this paper is as follow. We first briefly describe the background knowledge about model transformation and applying design patterns to model transformation in the next section. The relationship between design patterns and non-functional issues is depicted in section 3, and the section 4 provides the transformation rules. A meeting scheduling system is provided as an example to illustrate our approach in section 5, and some concluding remarks are given in section 6.

2. Background

A number of researches in the area of model transformation and pattern based model transformation have made their marks on our goal-driven pattern-based model transformation method.

2.1. Model transformation approaches

Graph transformation. Varró et al [16] describe a system for model transformation based on Graph Transformations. This style of transformation is based on the application of an ordered set of rules. Operators available include transitive closure, and repeated application. Rules identify sub-graphs which define before and after states, and may refer to source and target model elements and introduce associations between.

Relational transformation. The basic idea is to state the source and target element type of a relation and specify it using constraints. Gerber et al. [9] explore the application of logic programming to implement transformations.

Model transformation through high-level programming language. Implementing a model transformation can be carried out by using a general programming language with a specific API. Indeed, nowadays, repositories such as dMOF [4] can be used to save models and metamodels that are MOF compliant. These tools allow an API to be generated and its basic implementation for each contained metamodels.

Model transformation based on transformation metamodel. Engineers build models to better understand the systems that are being developed. In a similar way, to understand existing models we may provide models of these as well. This activity is called meta-modeling. K. Duddy et al. [5] defined a transformation metamodel of which instances will be input parameters of a generator, which transforms a model into a program that implements the transformation process described by this model.

2.2. Pattern-based model transformation approaches

Work in a number of fields has made their marks on the pattern-based approach. These researches are organized into following categories: automatic application of design patterns, quality improvement by design patterns, pattern-based development methodology, and others. The first category of researches [1, 12] focuses on automatic pattern application and ignores the usage of design patterns. The second category of researches [3, 11, 13] focuses on how to restructure a legacy system to be more qualified for the assistance of pattern-based transformation skills. The third category [2, 8] attempts to providing a methodology to serve as a bridge between users' requirements and design pattern technology. They put efforts on analysis of user needs rather than automation of pattern application.

3. Design patterns and non-functional requirements

Non-functional requirements are not easy to handle because they are subjective (they can be interpreted differently by different people), relative (their importance is depending on the system being considered) and interacting (their achievement may impair or help that of others) [6]. Design patterns [7] provide a possible way to deal with non-functional requirements since they provide solutions to satisfy functional requirements as well as better solutions to meet non-functional requirements. In particular, besides providing a basic, functional solution to a problem, a design pattern offers a qualified, non-functional improvement to that solution. For example, considering the original intent described in *Observer* design pattern:

> *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.*

By elaborating the intent, we can understand the design pattern is designed to resolve the communication between a subject objects and its related observer objects. Viewing from the functional aspect, it requires the subject to notify all observers when the subject changes its state. Viewing from the non-functional aspect, it requires the notification should work automatically without knowing types of observers. In other words, Observer design pattern has a FR-intent (functional requirement intent) to address functional problems and an NFR-intent (nonfunctional requirement intent) to improve non-functional quality requirements. We thus transform a design pattern's

intent into functional-intent (FR-intent) and non-functional intent (NFR-intent) to highlight the quality contribution of the pattern.

The structure distributes the object model to achieve the goal described in the intent property. With respective to the FR-intent and NFR-intent, FR-structure and NFR-structure are developed respectively. The major difference of NFR-structure to FR-structure is it applies object techniques to resolve problems. These techniques, such as polymorphism, abstraction, encapsulation, delegation, dynamic binding and inheritance are keys to make object-oriented system more reusable, extensible and maintainable. More details about the relationships between intent and structure of FR and NFR please see [10].

4. Transformation Rule Schema

Each refinement process is based on the application of a design pattern. A transformation is described graphically by a schema called transformation rule schema. A transformation rule schema is parameterized by model elements to be specified by the designer, and is composed of two compartments. The first compartment describes the source model of the design while the second compartment shows its corresponding target model after application of a design pattern. Fig. 2 shows the transformation rule schema that we have defined for the Observer pattern.

After establishing the goals hierarchy for obtaining alternative models and constructing stable kernel and alternatives, the designer must specify the analysis model based on the goals hierarchy in an incremental fashion. Initially a goal in goals hierarchy is chosen that will serve as a start point for the design pattern transformation applying under our proposed approach.

Rule-name: Observer-001
Purpose: The subject object can notify the interested objects without specifying their concrete classes
Category: Non-functional requirements (extensibility, consistency)
Using pattern: Observer
Preconditions:
When a change to one object requires changing others; or when an object should be able to notify other objects without having to know the identify of these objects.
Transformation actions:
1. (*Abstraction*) For each subject and observer class creates a new superclass (abstract class)
2. (*Inheritance*) Build the inheritance relation between the superclass and its concrete class.
3. (*Operation addition*) Insert the attach() and detach() operations into the subject's super class.
4. (*Association change*) Change the association between subject and observers
   4.1 Make an association between subject's superclass to observer's superclass
   4.2 Delete the association between subject to observer

Fig. 2: Transformation rule schema of *Observer* pattern

According to the functional and non-functional aspects of the chosen goal, we can match one or more design patterns to deal with this goal's non-functional requirement. Designer can choose one of these design patterns suitable to match the non-functional requirement, and apply transformation rules defined with this design pattern to satisfy the non-functional goal.

5. Meeting scheduling example

In this section, we use an example - meeting scheduling system to demonstrate the idea of our approach. Goals are identified based on the proposed verb-based classification scheme. In this example, we have identified five goals and formed a goal-driven use case model (see Fig. 3).

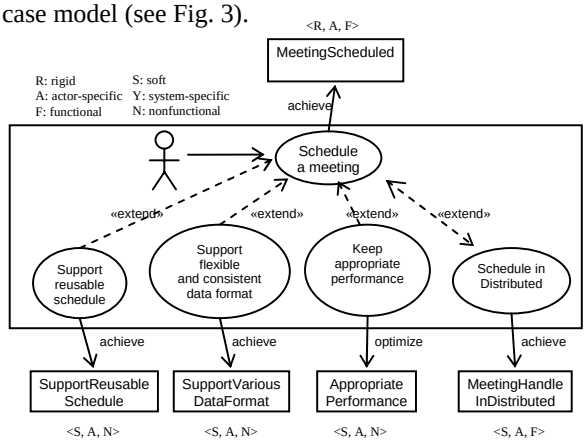

Fig. 3: Goal-driven use case model for the meeting scheduling system

- Schedule: [initiator, meeting date and location, initiator, participants, ϕ, rigid] (G_{MS})
- Handle: [system, plan meetings, initiator, ϕ, in distributed, rigid] (G_{MHID})
- Support: [initiator, meeting date and location, initiator, participants, reusable schedule, soft] (G_{SRS})
- Support: [system, meeting date, initiator, participants, support flexible and consistent date format , soft] (G_{SFC})
- Provide: [system, performance, ϕ, ϕ, an appropriate level, soft] (G_{AP})

An initial analysis model is obtained from the goals hierarchy and the construction of the design model is proceeded based on stable kernel in an incremental fashion from G_{MS} to G_{SFC}. Since the G_{MHID} represents the system can be scheduled and handled in distributed manner, the designer chooses the remote proxy pattern to support the distributed management. By considering the G_{SRS} and G_{SFC} to realize the goal *SupportReusableSchedule* and the goal *SupportVariousDataFormat*, the abstract factory pattern and the observer pattern are chosen to enhance the reusable scheduling and support various data format, respectively.

The G_{SFC} represents the meeting scheduling system could support various data format and keep consistence between different data views, i.e., a change on one data view must make the same change on the others. To resolve the inconsistency problem,

the Observer design pattern is recommended and used. *Observer* pattern is used to resolve the inconsistency problem between a set of objects (said observers) which have a common interest on subject. By requiring the observers to register on the subject before operating, observers can be notified for keeping consistency whenever the subject changes its status.

Fig. 4 represents the final design model after applying the design patterns through our proposed approach.

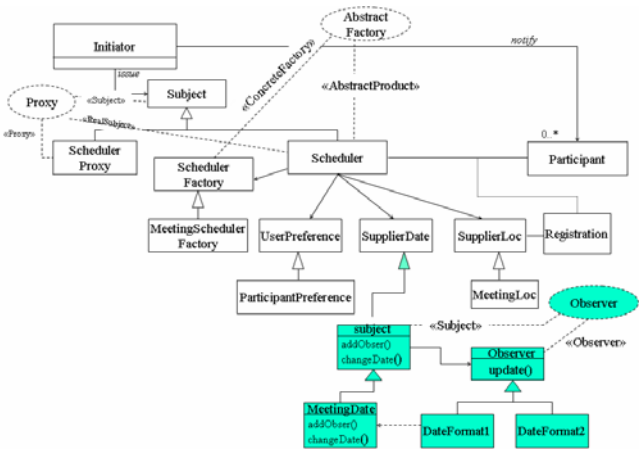

Fig. 4: Goals considered: {G_{MS}, G_{MHID}, G_{SRS}, G_{SFC}} after applying *Observer* pattern

6. Conclusion

In this paper, we propose an approach to providing a goal-driven method to transform the analysis model into design model for dealing with the software quality issues based on design patterns. With the aid of design patterns, in our approach, a goal can be achieved by a well-specified transformation rules, recovered by a proven solution, and enhanced by a known skill.

Acknowledge

This work was supported by the Ministry of Economic Affair (Taiwan) under grant 95-EC-17-A-02-S1-029

References

[1] F. Budinsky, M.Finie, J.Vlisdes, and P.Yu. Automatic code genration from design patterns. *Object Technology*, 35(2):172-191, 1996.

[2] L. Chung, K. Cooper, and A. Yi. Developing adaptable software architectures using design patterns: an NFR approach. Computer Standards and Interfaces, 23:253-260, 2003.

[3] M.O. Cinneide. Automated refactoring to introduce design patterns. In *Proceedings of the International Conference on Software Engineering*, 2000.

[4] dMOF user guide (1.1 release). Distributed Systems Technology Center (DSTC), Australia, July 2000.

[5] K. Duddy, A. Gerber, M. Lawley, K. Raymond, and J. Steel. Declarative transformation for object-oriented models. *Transformation of Knowledge*, Information and Data: Theory and Application, Idea Group, 2004.

[6] C. Ebert. Putting requirement management into praxis: dealing with nonfunctional requirements. *Information and Software Technology*, 40:175-185, 1998.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of reusable object-oriented software, Reading, MA: Addison-Wesley, 1994.

[8] E. Gross and E. Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 6:18-36, 2001.

[9] A. Gerber, M. Lawley, K. Raymond, J. Steel, A. Wood. Transformation: The Missing Link of MDA, In A. Corradini, H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.): *Graph Transformation: First International Conference (ICGT 2002)*, 2002.

[10] N.-L. Hsueh and W.-H. Shen. Handling Nonfunctional and Conflicting Requirements with Design Patterns. In *the proceedings of the 11th Asia-Pacific Software Engineering Conference*, Page(s):608 – 615, 2004.

[11] B. Huston. The efforts of design pattern application on metric scores. *The Journal of Systems and Software*, 58:261-269, 2001.

[12] S. Jeon, J. Lee, and D. Bae. An automated refactoring approach to design pattern-based program transformations in Java programs. In *the proceedings of the Ninth Asia Pacific Conference of Software Engineering*, pages 337-345, 2002.

[13] L. Khriss, R.K. Keller, and I. Hamid. Pattern-based refinement schemas for design knowledge transfer. *Knowledge-Based Systems*, 13:403-415, 2000.

[14] J. Lee and N.L Xue. July/August 1999. Analyzing user requirements by use cases: A goal-driven approach. *IEEE Software*, 16 (4):92-101.

[15] J. Lee, N.L. Xue, K.H. Hsu and S.J.H. Yang. 1999. Modeling imprecise requirements with fuzzy objects. *Information Sciences*, 118:101-119.

[16] D.V.G. Varró and A. Pataricza. Designing the automatic transformation of visual languages. *Journal of Science and Computer Programming*, 44: 205-227, 2002.