Can Abstraction Be Taught?
Refactoring-based Abstraction Learning

Naoyasu Ubayashi, Yasutaka Kamei and Ryosuke Sato
Kyushu University, Fukuoka, Japan

Keywords: Abstraction, Refinement, Abstraction-aware Refactoring.

Abstract: Can the notion of abstraction be taught to students? It is a very difficult question. Abstraction plays an important role in software development. This paper shows that refactoring is effective for students to learn the notion of abstraction. We focus on design abstraction, because it is one of the crucial parts in teaching the essence of software engineering. To explore for a well-balanced separation of concerns between design and code, it is not avoidable to go back and forth between them. To help a student find an appropriate abstraction level, we introduce abstraction-aware refactoring patterns consisting of *MoveM2C* (Move concerns from Model to Code) and *CopyC2M* (Copy concerns from Code to Model). The patterns enable a student to refine abstraction while preserving not only external functionality but also traceability between design and code.

1 INTRODUCTION

Can the notion of abstraction be effectively taught to students? It is a very difficult question. Abstraction plays an important role in software development. J. Kramer not only claims that abstraction is crucial for computing professionals but also raises questions (Kramer, 2007): *Why is it that some software engineers and computer scientists are able to produce clear, elegant designs and programs, while others cannot? Is it possible to improve these skills through education and training?* His hypothesis is that *critical to these questions is the notion of abstraction*. Many researchers and developers agree that abstraction is one of the most important skills in software engineering. However, there are different aspects of abstraction: abstraction in design specifications, abstraction in program modules, abstraction in software architectures, and so on. In this paper, we focus on abstraction between design and implementation, because it is one of the crucial parts in teaching the essence of software engineering to university students. In many universities, programming is taught first and after that software design is taught. Most students feel that there is a big gap between software design and programming in terms of abstraction.

This paper shows that refactoring crosscutting over design and code is effective for ordinary students to learn the notion of abstraction. In (Batini and Viscusi, 2016), the etymology of abstraction is shown as follows: *(14c.) from Latin abstractionem (nominative abstractio), noun of action from past participle stem of abstrahere "drag away, pull away, divert"*. From this etymology, abstraction can be defined as *removal of details*. There are two different abstractions: *abstraction by forgetting (details)* and *abstraction by hiding*. Class hierarchy is an example of the former type. On the other hand, the latter type is crucial in terms of abstraction between design and code, because design is a specification of software architecture (Shaw and Garlan, 1994) hiding detail implementation.

In this paper, we introduce a set of patterns for abstraction-aware refactoring. The patterns consist of *MoveM2C* (Move concerns from Model to Code) and *CopyC2M* (Copy concerns from Code to Model). In our educational experience, students often fail to make a model at the first time, because they cannot understand to what extent the model should be detailed. Some students make a very detailed model equal to the program code. Such a model is hard to maintain, because it has to be modified whenever the code is revised. Some students make a too abstract model containing only few model elements and cannot understand how to relate the model to the code. They have the knowledge of UML notations but do not understand in depth what is abstraction. On the other hand, it is relatively easy for them to write a program, although abstraction skills in programming might be insufficient. Our teaching objective is to

Ubayashi, N., Kamei, Y. and Sato, R.
Can Abstraction Be Taught? Refactoring-based Abstraction Learning.
DOI: 10.5220/0006604804290437
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 429-437
ISBN: 978-989-758-283-7
Copyright © 2018 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

help students understand how to remove details at the beginning of learning modeling. Traditional refactoring (Fowler, 1999) improves the code structures while preserving the external functionality. Our refactoring patterns refine abstraction while preserving not only external functionality but also traceability between design and code. Traceability is important for students to understand how a model is related to the code in software development. Although it is preferable to firmly separate design from its implementation (or hide non-essential implementation details), this separation is not easy for many students because an abstraction level—*How much should a design be more abstract than code?*—tends to change during the progress of software development. In general, an important decision on software architecture is made at the design phase and a decision on the detailed program structure concerning API usages, variables, and methods is made at the coding phase. However, this distinction is relative and vague in many cases. R. N. Taylor et al. pointed out the need for adequate support for fluidly moving between design and coding tasks (Taylor and Hoek, 2007). Because of *fluid moving*, abstraction level may fluidly change as a result of reconsidering the balance between design and its implementation—which concern should be described in design and which concern should be written in code. Our refactoring patterns supports *fluid abstraction*, a design approach in which appropriate abstraction can be captured by the convergence of fluid moving and removing details. Why do we use the word *fluid abstraction*? The reason is that abstraction in design and coding phases should not be absolutely given by a teacher but be flexible for a student to seek the best combination of design and code.

We evaluated the effectiveness of the proposed refactoring patterns by applying them to a system developed in our university's PBL (Project-Based Learning) class. This experiment gave us the interesting knowledge about teaching software development in terms of abstraction. We addressed three research questions: RQ1) How much value does an abstraction level become in an educational project? ; RQ2) What becomes the trigger of applying the refactoring patterns? ; and RQ3) Does an abstraction level finally converge to a certain value by applying the refactoring patterns? For RQ1, the abstraction level l ($0 \leq l \leq 1$) was about around 0.5 in the educational system development. For RQ2, we could extract two bad smells triggering abstraction-aware refactoring: *"abnormal abstraction level"* and *"inconsistency between design and code."* For RQ3, all design models converged to the similar abstraction level. We consider that the existence of a converged abstraction level is important for a student to capture an abstraction skill, because he or she can understand that there is an abstraction structure in a real system.

This paper is structured as follows. The technical background needed to understand our approach is briefly explained in Section 2. Abstraction-aware refactoring patterns are introduced in Section 3. In Section 4, we show *iArch*, a support tool for applying the refactoring patterns. In Section 5, we show the evaluation results in our university's PBL class. In Section 6, we show related work. Concluding remarks are provided in Section 7.

2 BACKGROUND

In this section, we introduce the technical background of the abstraction-aware refactoring patterns. Our approach is based on an interface mechanism that represents a contract between design and code. This interface is called *Archface* (Architectural Interface) (Ubayashi et al., 2014). A student has to define this interface that enforces the student to decide which concern should be included in both a design model and code. The student has to remove the non-essential details from the design model by not specifying them in the interface.

2.1 Archface

Archface exposes architectural points shared between design and code. These points termed *archpoints* have to be modeled as design points in a UML model and have to be implemented as program points in its code. *Archface* is a contract that should be guaranteed between design and code in terms of abstraction. An abstraction level is determined by selecting archpoints that should be shared between design and code. There are two kinds of type checking: type check between *Archface* and a UML model; and type check between *Archface* and code. If both type checks are correct, a design model is traceable to the code in terms of a defined abstraction level.

Table 1 shows design points, program points, and archpoints. It is easy for a student to understand which archpoint corresponds to a model element (design point) in a design model or a program snippet (program point). We limited archpoints to basic points that can be easily understood by students, although more complex archpoints can be introduced (e.g., data flow archpoints). There are several merits in using the notion of *points*. One of the essential merits is that points are countable and an abstraction level can be easily calculated. There are two types of

Table 1: Design Points, Program Points, and Archpoints.

Design point (UML2 metamodel)	Program point (Java)	Archpoint
`Class`	class	a_class
`Operation`	method	a_method
`Property`	field	a_field
`Message MessageEnd (sendEvent)`	method call	a_mcall
`Message MessageEnd (receiveEvent)`	method exec	a_mexec

archpoints: structural archpoints based on class structures and behavioral archpoints based on LTS (Labelled Transition Systems). Both structural and behavioral aspects are essential in learning modeling.

Archface, which supports *component-and-connector* architecture (Allen and Garlan, 1994), consists of two kinds of interface: *component* and *connector*. The former exposes archpoints and the latter defines how to coordinate them. An *Archface* definition of the *Observer* pattern is shown in List 1.

```
[List 1] -- Java & FSP Syntax
01: interface component cSubject {
02:   public void addObserver(Observer);
03:   public void removeObserver(Observer);
04:   public String getState();
05:   public void setState(String);
06: }
07:
08: interface component cObserver {
09:   public void update();
10: }
11:
12: interface connector cObserverPattern
13:             (cSubject, cObserver){
14:   cSubject = (cSubject.setState->cObserver.update
15:             ->cSubject.getState->cSubject);
16:   cObserver = (cObserver.update->cSubject.getState
17:             ->cObserver);
18: }
```

The structural aspect is described as a Java-like interface and the behavioral aspect is specified using process algebra. That is, message interactions are described based on FSP (Finite State Processes) (Magee and Kramer, 2006) whose specifications generate finite LTS. The notation $\rightarrow$ indicates a control flow represented by a method call. An interaction between a subject and observers is defined in line 14-17. A message is sent from a subject to an observer by `cObserver.update` in line 14 and 16. The `cSubject` is specified as a process that repeatedly receives `setState`, sends `update` to the `cObserver`, and receives `getState`. In the same way, the `cObserver` is specified as a process that repeatedly receives `update` and sends `getState` to the `cSubject`.

2.2 Abstraction-aware Compiler

The conformance to *Archface* can be checked by a type system taking into account not only programs but also design models. Type checking is performed by verifying whether or not a design point (program point) corresponding to an archpoint exists in a design model (program) while satisfying constraints among design points (program points) (e.g., the order of message sequences). Although traditional types are structural, *Archface* is based on archpoints including behavior. The reason is because a design model imposes structural or behavioral architectural constraints on a program. Our type checker verifies the simulation relation between a design model and its code via FSP descriptions in *Archface*. Fixing inter-model inconsistency is an important problem (Egyed et al., 2008) when students understand how a diagram is related to other diagrams and which role each diagram has. Our compiler can verify inconsistency not only between a model and code but also between models.

2.3 Abstraction Level

An abstraction level—*How much should a design model be more abstract than its code?*—is determined by selecting archpoints that should be shared between design and code. The *abstraction ratio* is a metric for measuring an abstraction level. The value of this metric is $1 - \#ArchPoint / \#ProgramPoint$. $\#ArchPoint$ and $\#ProgramPoint$ are the number of archpoints and program points, respectively. We provided a simple metric for a student to understand abstraction intuitively and quantitatively. $\#DesignPoint$, the number of design points, can be larger than $\#ArchPoint$ when there are design concerns that are not reflected into the code. On the other hand, $\#ArchPoint$ indicates the number of design points that should be reflected to the code. The value of this metric ranges from 0 to 1. A large value close to 1 indicates that the abstraction level is high. A small value close to 0 indicates that the abstraction level is low. The number of archpoints and program points is automatically calculated from the *Archface* definitions by our support tool *iArch* shown in Section 4. In case of $Ratio = 0$, all archpoints are mapped to the corresponding program points. This case indicates that a model is equal to the code. In case of $Ratio = 1$, there are no design descriptions corresponding to a program. There are no archpoints. A software artifact that does not contain design descriptions is an example of this case.

3 REFACTORING PATTERNS

In this section, we introduce the abstraction-aware refactoring patterns that help a student explore an appropriate abstraction level ranging between 0 and 1. The refactoring proposed by us refines an abstraction level while preserving not only external functionality

Table 2: Abstraction-aware Refactoring Patterns.

Category	Pattern		Abbreviation	Explanation	Abstraction
MoveM2C	Structural	①Remove Class	RmCs	Remove a non-important class.	↗
		②Remove Sub Class	RmSubCs	Remove a non-important sub class.	↗
		③Remove Library Class	RmLibCs	Remove a non-important API class.	↗
		④Remove Method	RmMs	Remove a non-important method.	↗
		⑤Remove Field	RmFd	Remove a non-important field.	↗
	Behavioral	⑥Remove Message	RmMsg	Remove non-important message	↗
		⑦Remove API Call	RmAPI	Remove a non-important API call.	↗
		⑧Remove Object	RmObj	Remove a non-important object.	↗
CopyC2M	Structural	⑨Add Class	AddCs	Add an important class existing in the code to its design model.	↘
		⑩Add Sub Class	AddSubCs	Add an important sub class existing in the code to its design model.	↘
		⑩Add Library Class	AddLibCs	Add an important API class existing in the code to its design model.	↘
		⑫Add Method	AddMs	Add an important method existing in the code to its design model.	↘
		⑬Add Field	AddFd	Add an important field existing in the code to its design model.	↘
	Behavioral	⑭Add Message	AddMsg	Add an important message send/receive (method call) existing in the code to its design model.	↘
		⑮Add API Call	AddAPI	Add an important API call in the code to its design model.	↘
		⑯Add Object	AddObj	Add an important object in the code to its design model.	↘

but also traceability between design and code. A student can fluidly go back and forth between design and code by using the refactoring patterns.

3.1 Pattern Catalog

Table 2 shows the pattern catalog composed of *MoveM2C* (Move concerns from Model to Code) and *CopyC2M* (Copy concerns from Code to Model).

A pattern in the *MoveM2C* category moves a design concern to a code concern. This pattern is applied to the situation in which a design model has to be changed frequently to reflect code change. It may be preferable to locate the concern to code. As another situation, this pattern can be temporarily applied if experimental coding is needed to find an appropriate design such as performance issues and correct API usages. After finding an adequate design, the *CopyC2M* pattern category should be applied to recover a design concern from its code. By applying the *MoveM2C* pattern, we can raise an abstraction level. The abstraction of a design model becomes an appropriate level and the design model becomes stable (not frequently modified). The *MoveM2C* category is divided into two kinds of patterns: one is the structural patterns for abstracting class structures and includes *RmCs*, *RmSubCs*, *RmLibCs*, *RmMs*, *RmFd* patterns. Another is the behavioral patterns for abstracting message sequences and includes *RmMsg*, *RmAPI*, *RmObj* patterns. These patterns are related to each other. For example, if a message is removed from a sequence diagram (*RmMsg*), we have to remove the target object (*RmObj*) and associated message sequences (*RmMsg*). Moreover, we may have to remove a class instantiating the object (*RmCs*), its sub classes (*RmSubCs*), associated methods (*RmMs*), and fields (*RmFd*) from a class diagram. Although each pattern is simple, it is not easy to apply these patterns while preserving the consistency. Our support tool *iArch* can automate these tasks.

A pattern in the *CopyC2M* category copies a code concern to a design concern. This pattern, the reverse of the *MoveM2C* pattern, is applied to the situation in which a student wants to change a design model to reflect an important design concern that could not be captured in the early phase but can be obtained in the coding phase. By applying the *CopyC2M* pattern, we can lower an abstraction level. To perform the *CopyC2M* pattern efficiently, *iArch* supports the following: 1) a student selects a code region that should be reflected to the corresponding design model; 2) *iArch* shows the candidates of updated *Archface* descriptions; 3) the student selects most appropriate one from the candidates and modifies it if necessary; 4) a model editor in *iArch* updates a design model in order to reflect the updated *Archface*. If the design model already contains a design point related to a new archpoint defined in the updated *Archface*, the model editor simply reuses the design point.

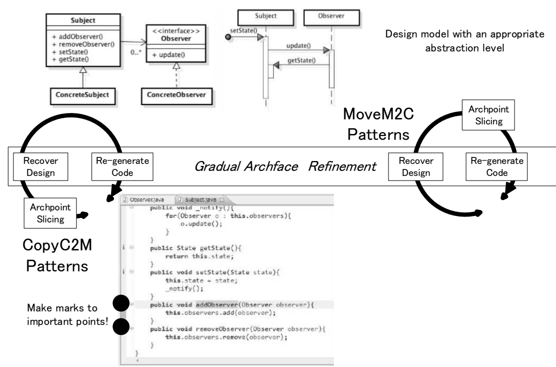

Figure 1: Gradual *Archface* Refinement.

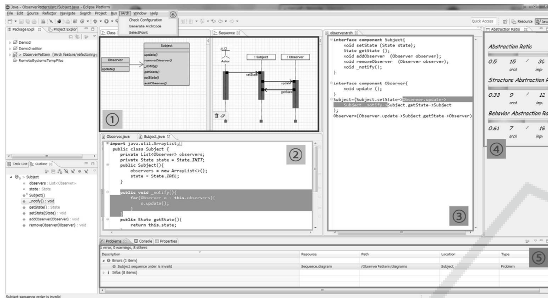

Figure 2: *iArch* IDE.

4 LEARNING ENVIRONMENT

Figure 1 illustrates the abstraction refinement process using *MoveM2C* and *CopyC2M*. This refinement process is supported by the *iArch* IDE (Ai et al., 2014) consisting of the followings: 1) model editor, 2) program editor, 3) *Archface* editor and generator, 4) abstraction metrics calculator, and 5) abstraction-aware verifying compiler. Figure 2 is a snapshot of *iArch* implemented as an Eclipse plug-in. 1) is class and sequence diagrams of the *Observer* pattern. 2) is a Java program implementing the *Observer* pattern. 3) is an *Archface* definition. We can automatically generate initial *Archface* descriptions from a design model. 4) shows the abstraction ratio of the *Observer* pattern. 5) is the output from the abstraction-aware verifying compiler. This compiler generates error messages if the traceability between a design model and code is violated. The refactoring support facility is newly added to the *iArch* IDE.

5 EXPERIMENT

We evaluated the effectiveness of our refactoring patterns by applying them to a system developed in our university class. Although this is a preliminary experiment and we cannot assume the generality of the result, it gives us an interesting intuition about teaching software development in terms of abstraction.

5.1 System Overview

A web-based online book-shop system was developed in a PBL, a class for master students to learn the methods and processes for developing a practical system. The artifacts of this system consist of UML models and programs written in Java and JavaScript. We used class diagrams, sequence diagrams, and Java programs as the target of this evaluation. This system consists of two main functions. One is a function for a shop user to search book information and check the user ranking. Another is a function for a shop clerk to manage book information. Table 3 shows the system configuration and its size (LOC: Lines of Code). The software modules consist of 1) definition of the relation among modules (abs); 2) access control for the database (asasecommon); 3) user service (web); and 4) clerk service (office).

5.2 Research Questions

In this experiment, we addressed the following three research questions: RQ1) How much value does an abstraction level become in an educational project?; RQ2) What becomes the trigger of applying the refactoring patterns?; and RQ3) Does an abstraction level finally converge to a certain value by applying the refactoring patterns?

RQ1 is intended to clarify whether or not some kind of tendency is seen about the value of an abstraction level. It is considered that the abstraction level varies according to a project theme and a target domain. However, there may be the similar tendency in a similar team or a similar PBL theme. Of course, we cannot definitely give a predictable value of the abstraction level, because it is just the evaluation by one project. RQ2 is intended to clarify bad smells that trigger a student to refactor a design model and its code. RQ3 is intended to measure the effect of applying the refactoring patterns.

5.3 Experiment Results

RQ1: Table 4 shows the abstraction level of the online book shop system. The value was about around 0.5 in the system development of the educational purpose. This value may be slightly lower than that of the system development in companies. We consider that the value is appropriate because one of the purposes of this educational PBL is to help students un-

Table 3: Project Size of an Online Book Shop System.

Module Name	Number of Classes	Number of Class Diagrams	Number of Sequence Diagrams	LOC
abs	23	1	0	411
asasecommon	26	1	0	1393
web	9	1	2	244
office	20	1	4	438
UI	—	—	—	3170

Table 4: Initial Abstraction Level of the System.

ID	Use Case (*)	Archpoint	Program Point	Inconsistency	Abstraction Ratio
u1	User: Check User Ranking	9	22	√	0.59
u2	User: Search Book Titles	16	24		0.33
u3	Clerk: Search/Update/Delete Publisher Information	31	76	√	0.59
u4	Clerk: Register Publisher Information	12	24	√	0.5
u5	Clerk: Search/Update/Delete Book Information	37	82	√	0.55
u6	Clerk: Register Book Information	12	24	√	0.5

* Each use case is modeled using a sequence diagram.

Table 5: Bad Smells and Refactoring Patterns to be Applied.

ID	Bad Smell	Reason	Refactoring Patterns
u1	B	There is an inconsistency between a sequence diagram and code, because the usage of API is changed.	*RmAPI*, *RmObj*, *AddObj*, *AddMsg*
u2	A	It is not necessary to distinguish the brief search and the detailed search.	*RmMsg*
u3	B	A part of functions are not modeled in a sequence diagram.	*AddObj*, *AddMsg*, *RmMsg*
u4	B	Same as above.	*AddObj*, *AddMsg*, *RmMsg*
u5	B	Same as above.	*AddObj*, *AddMsg*, *RmMsg*
u6	B	Same as above.	*AddObj*, *AddMsg*, *RmMsg*

A: Abnormal Abstraction Level, B: Inconsistency between Design and Code

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 6: Change of Abstraction Level at Each Refactoring Step.

ID	Refactoring Step		
	1: Recover Traceability between Design and Code	2: Apply *CopyC2M*	3: Apply *MoveM2C*
u1	*RmAPI*, *RmObj* / +0.09	*AddObj*, *AddMsg* / -0.05	—
u2	—	—	*RmMsg* / +0.25
u3	—	*AddObj*, *AddMsg* / -0.12	*RmMsg* / +0.15
u4	—	*AddObj*, *AddMsg* / -0.12	*RmMsg* / +0.16
u5	—	*AddObj*, *AddMsg* / -0.11	*RmMsg* / +0.14
u6	—	*AddObj*, *AddMsg* / -0.12	*RmMsg* / +0.16

Legend: Applied Patterns / Change of Abstraction Level (Difference Before and After Refactoring)

derstand the traceability between a design model and the code in terms of abstraction. It is easy for a student to understand abstraction when a design model is not too abstract and is relatively close to its code. On the other hand, the student may fail to capture the abstraction skills if the abstraction level is larger than 0.5. We consider that it is effective to suggest to students that they should make a design model with the abstraction level 0.5 as the first learning step. Students will find more appropriate abstraction level after they can understand the meaning of 0.5.

RQ2: In this experiment, students in the PBL class extracted two kinds of bad smells that triggered the application of the abstraction-aware refactoring patterns: 1) Abnormal abstraction level; and 2) Inconsistency between design and code. 1) indicates that refactoring is needed if a class or a method has an exceptional abstraction ratio comparing other classes or methods. 2) indicates the situation in which traceability is not preserved between artifacts. Someone may consider why 2) is a bad smell, because the traceability has to be maintained in software development. However, many student projects cannot necessarily preserve the traceability, because it has to be maintained by hand in most cases. Current MDD tools do not provide the functions for practical traceability checking. If traceability is not maintained, design models tend to evolve without taking into account the code. As a result, the abstraction level of the design model may become inadequate. Table 5 shows bad smells and refactoring patterns to be applied. In Table 4, the abstraction level of the use case `u2` is 0.33. We cannot determine whether this value is appropriate or not by checking only this value, al-

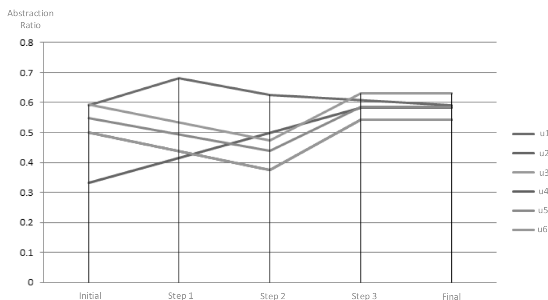

Figure 3: Change History of Abstraction Levels.

though one might consider the value is relatively low. However, this value is exceptionally low comparing to the abstraction levels of other use cases whose functionality is similar to that of the use case u2. There was a possibility of a bad smell. We checked the sequence diagram representing the use case u2 in detail and found that a function “Search book information” was divided into two use cases “brief search” and “detailed search”. In this case, it is desirable to refactor the sequence diagram representing u2 by applying the *RmMsg* pattern. These bad smells help a student capture the practical skills for abstraction.

RQ3: In this PBL class, the students performed three refactoring steps: 1) recover the traceability link between a design model and code; 2) apply the *CopyC2M* pattern; and 3) apply the *MoveM2C* pattern as shown in Table 6 and Figure 3. All sequence diagrams converged to the similar abstraction level. The gap between maximum and minimum abstraction level is changed from 0.26 (= 0.59-0.33) to 0.09 (= 0.63-0.54). Our refactoring patterns help a student refine and explore an appropriate abstraction level. The student can understand that he or she can obtain a simple and beautiful software structure by seeking an appropriate abstraction level.

6 RELATED WORK

Someone might wonder what is a distinction between our approach and a view supported by many modeling tools that can hide a portion of a model. If a developer makes a detailed model equal to the code, an abstract model can be obtained from the original model without using our approach. However, a student cannot get a skill for making an abstract model but write code just using modeling notations. Moreover, in practical development, it is not realistic to make a code-level model, because the model is not only useless but also difficult to maintain.

As claimed in this paper, traceability between design and code plays an important role in teaching abstraction, because an abstract design model cannot exist without considering the relation to its program implementation. There are several studies on traceability. T. D’Hondt et al. introduced LMP (Logic-Meta Programming) to enforce the synchronization between object-oriented design and code (D’Hondt et al., 2001). H. Bagheri et al. showed a way for automated formal derivation of style-specific architectures (Bagheri et al., 2010). Y. Zheng and R. N. Taylor proposed 1.x-way architecture-implementation mapping (Zheng and Taylor, 2012) for deep separation of generated and non-generated code. JaMoPP[1], a set of plug-ins for parsing Java code into models based on EMF (Eclipse Modeling Framework), bridges the gap between modeling and programming. MoDisco[2] is a framework to develop model-driven tools supporting software modernization. Both JaMoPP and MoDisco provide reverse engineering facilities. In the past, many ADLs (Architecture Description Languages) have been proposed to describe an architectural design model at a high abstraction level. R. Allen and D. Garlan proposed *Wright* (Allen and Garlan, 1994) to formalize the *component-and-connector* architecture. J. Aldrich et al. proposed *ArchJava* (Aldrich et al., 2002), an extension of Java. ArchJava unifies architecture and implementation, ensuring that the implementation conforms to architectural constraints. *Umple*[3] supports the notion of model-oriented programming that adds modeling features derived from UML to object-oriented languages such as Java. Using *ArchJava* or *Umple*, we can merge modeling with programming. Cassou et al. explored the design space between abstract and concrete component interaction specifications (Cassou et al., 2011).

However, these approaches do not support fluid moving while preserving abstraction. They take the policy that separation of concerns between design and code should be firmly determined at the initial design phase. However, we often observe fluid moving and frequent change of abstraction in educational projects as repeatedly claimed in this paper, because an ordinary student cannot create a well-abstract design at the initial phase. It is necessary to provide a method for supporting fluid moving in the light of abstraction. Our proposal is one of the solutions for dealing with this educational challenge.

Automated verification tools supporting formal specifications are useful for students to capture their abstraction skills, because the repeated cycle of trial-and-error development gives the students opportunities of rethinking the abstract specifications. For ex-

[1] http://www.jamopp.org/
[2] http://www.eclipse.org/MoDisco/
[3] http://cruise.eecs.uottawa.ca/umple/

ample, the Alloy analyzer (Jackson, 2006) is suitable for students to understand the essence of software abstractions. Our abstraction-aware verifying compiler helps a student understand the traceability between design and code in terms of abstraction.

7 CONCLUSIONS

In this paper, we showed that refactoring crosscutting over design and code is effective for ordinary students to learn software abstractions. However, many educational challenges still remain, because abstraction is a philosophical deep concept. Our definition of abstraction (*removal of details*) is based on archpoints, a simple mechanism for not only representing abstraction but also realizing traceability. However, abstractions existing in a real software development project are not limited to the concepts in which design points and program points are mapped each other. Nevertheless, we consider our approach is effective for educational purposes, because it can integrate many important notions such as definition of abstraction level, traceability check, and refactoring both design models and code.

Although this paper focused on the educational aspect in university, we expect that our approach can be applied to leverage the modeling skills of primary engineers. The lack of proper skills for abstraction is a real problem in industry. The first author of this paper worked in industry as a software engineer for twenty years before moving to academia. To tell the truth, this research is motivated by the author's experience in industry. We believe that a systematic method for improving abstraction skills is one of the most important issues both in academia and industry. Unfortunately, such a method does not exist yet. Our proposal is a first step towards method development for improving abstraction skills.

ACKNOWLEDGMENTS

We thank Di Ai, Peiyuan Li, Yuning Li, and Zhongxiao Guo for their great contributions. They were students of our university. They implemented the *iArch* IDE. Di Ai assisted the evaluation shown in Section 5. This work was supported by JSPS KAKENHI Grant Numbers JP26240007, JP25540025.

REFERENCES

Ai, D., Ubayashi, N., Li, P., Hosoai, S., and Kamei, Y. (2014). iarch: An ide for supporting abstraction-aware design traceability. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014), pp.442-447*.

Aldrich, J., Chambers, C., and Notkin, D. (2002). Archjava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), pp.187-197*.

Allen, R. and Garlan, D. (1994). Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94), pp.71-80*.

Bagheri, H., Song, Y., and Sullivan, K. J. (2010). Architectural style as an independent variable. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010), pp.159-162*.

Batini, C. and Viscusi, G. (2016). Er2016 - tutorial on abstractions in conceptual modelling and surroundings.

Cassou, D., Balland, E., Consel, C., and Lawall, J. (2011). Leveraging software architectures to guide and verify the development of sense/compute/control applications,. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), pp.431-440*.

D'Hondt, T., Volder, K. D., Mens, K., and Wuyts, R. (2001). Co-evolution of object-oriented software design and implementation. In *Software Architectures and Component Technology, pp.207-224*. Kluwer Academic Publishers.

Egyed, A., Letier, E., and Finkelstein, A. (2008). Generating and evaluating choices for fixing inconsistencies in uml design models. In *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE 2008), pp.99-108*.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Jackson, D. (2006). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

Kramer, J. (2007). Is abstraction the key to computing? In *Communications of the ACM, Vol. 50 Issue 4, pp.36-42*.

Magee, J. and Kramer, J. (2006). *Concurrency: State Models and Java Programs*. Wiley.

Shaw, M. and Garlan, D. (1994). Characteristics of higher level languages for software architecture. In *Technical Report, CMU-CS-94-210*. Carnegie Mellon University.

Taylor, R. N. and Hoek, A. (2007). Software design and architecture –the once and future focus of software engineering. In *Proceedings of 2007 Future of Software Engineering (FOSE 2007), pp.226-243*.

Ubayashi, N., Ai, D., Li, P., Li, Y., Hosoai, S., and Kamei, Y. (2014). Abstraction-aware verifying compiler for yet another mdd. In *Proceedings of the 29th Interna-*

tional Conference on Automated Software Engineering (ASE 2014), New Ideas Paper, pp.557-562.

Zheng, Y. and Taylor, R. N. (2012). Enhancing architecture-implementation conformance with change management and support for behavioral mapping. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), pp.628-638*.