iArch: An IDE for Supporting Fluid Abstraction

Di Ai
Kyushu University
Fukuoka, Japan
aidi@posl.ait.kyushu-u.ac.jp

Naoyasu Ubayashi
Kyushu University
Fukuoka, Japan
ubayashi@acm.org

Peiyuan Li
Kyushu University
Fukuoka, Japan
lipeiyuan@posl.ait.kyushu-u.ac.jp

Daisuke Yamamoto
Kyushu University
Fukuoka, Japan
yamamoto@posl.ait.kyushu-u.ac.jp

Yu Ning Li
Kyushu University
Fukuoka, Japan
liyuning@posl.ait.kyushu-u.ac.jp

Shintaro Hosoai
Kyushu University
Fukuoka, Japan
hosoai@qito.kyushu-u.ac.jp

Yasutaka Kamei
Kyushu University
Fukuoka, Japan
kamei@ait.kyushu-u.ac.jp

Abstract

Abstraction plays an important role in software development. Although it is preferable to firmly separate design from its implementation, this separation is not easy because an abstraction level tends to change during the progress of software development. It is not avoidable to fluidly go back and forth between design and implementation. An abstraction level of a design specification may change as a result of reconsidering the balance between design and code—which concern should be described in design and which concern should be written in code. The *iArch* IDE (Integrated Development Environment) supports the notion of *fluid abstraction*, a design approach in which an appropriate abstraction level can be captured by the convergence of fluid moving between design and implementation.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Languages

General Terms Design, Languages

Keywords Architecture, Interface, Abstraction

1. Introduction

Abstraction plays an important role in software development. J. Kramer not only claims that abstraction is crucial for computing professionals but also raises questions [8]: *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.*

It is important to rethink abstraction in the light of the relation between design and implementation. We claim that interface mechanisms between them play an important role in exploring appropriate abstraction. Design is an abstract specification of software architecture [2, 3] and code implements its design. Although it is preferable to firmly separate design from its implementation, this separation is not easy because an abstraction level—*How much should be a design 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. For example, API usages may be taken into account at the design phase if a developer considers that the usages can affect the design structure. Moreover, the importance of some classes and methods might be recognized at the coding phase even if they should be extracted at the design phase, because developers cannot always grasp all of the design concerns. It is not avoidable to go back and forth between design and implementation. As one of the important research directions in the field of software design and architecture, R. N. Taylor et al. pointed out the need for adequate support for fluidly moving between design and coding tasks [9]. Because of *fluid moving*, the abstraction level of a design 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. Appropriate abstraction determines proper separation of concerns between design and implementation.

The *iArch* IDE (Integrated Development Environment) supports *fluid abstraction*, a design approach in which appropriate abstraction can be captured by the convergence of fluid moving. Why do we use the word *fluid abstraction*? The reason is that abstraction in

MODULARITY '14, April 22–26, 2014, Lugano, Switzerland.

ACM 978-1-4503-2773-2/14/04.
http://dx.doi.org/10.1145/2584469.2584477

Table 1. Design/Program Points and Archpoints

Diagram	Design point (UML2 metamodel)	Program point (Java)	Archpoint (Pointcut)
Class diagram (UML)	`Class`	class	a_class (`class`)
	`Operation`	method	a_method (`method`)
	`Property`	field	a_field (`field`)
Sequence diagram (UML)	`Message -sendEvent:MessageEnd`	method call	a_mcall (`call`)*
	`Message -receiveEvent:MessageEnd`	method exec	a_mexec (`execution`)*
	`Interaction`	(control flow)	(`cflow`)* **
Data flow	(`Property` def)	field set	a_def (`set`)*
	(`Property` use)	field get	a_use (`get`)*

*) AspectJ pointcut, **) Used with `call` or `execution` pointcut

design and coding phases should not be absolutely firm but be flexible to seek the best combination of design and code. The *iArch* IDE provides the followings: 1) an architectural interface mechanism for specifying abstraction, 2) metrics for measuring an abstraction level, and 3) a verification mechanism for checking the traceability between design and implementation while preserving a specified abstraction level. To achieve these goals, we adopt *Archface* [10–12], an architectural interface mechanism. *Archface* exposes a set of architectural points that should be shared between design and code. *Archface* can be considered as a kind of contract between design and code. We have to modify design or code when traceability is violated. This violation might be resolved if an abstraction level is changed by modifying *Archface*. Our approach is similar to CEGAR (Counter Example Guided Abstraction Refinement) [4], an automatic iterative abstraction refinement methodology in model checking.

This paper is structured as follows. *Archface*-Centric MDD (Model-Driven Development) is introduced in Section 2. The *iArch* IDE is illustrated in Section 3. Concluding remarks are provided in Section 4.

2. Archface-Centric MDD

In this section, we introduce *Archface*-Centric MDD, a foundation of *iArch*, by excerpting from our work [10–12].

2.1 Archface

Figure 1 shows the relation among design modules, program modules, and *Archface*, an interface for bridging them. In our approach, a design model is regarded as a design module, a first-class software module. The same *Archface* is modeled by a design model and is implemented by program code. If each type check is correct, a design model is traceable to the code. *Archface* plays a role of design interface for a design model. At the same time, *Archface* plays a role of program interface for code. *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. Class declarations, methods, and events such as *message send* are called design points.

Table 1 shows design/program points and archpoints. These points can be mapped each other. The idea of archpoints and their selection originates in AOP (Aspect-Oriented Programming) notion such as join points and pointcuts. Archpoints correspond to join points in AspectJ [7]. We focus on archpoints embedded in class and sequence diagrams, because structural and behavioral aspects of software architecture can be basically represented using these diagrams. *Archface* conceptually includes the notion of traditional program interface, because a method definition can be interpreted as a provision of an archpoint selected by an `execution` pointcut.

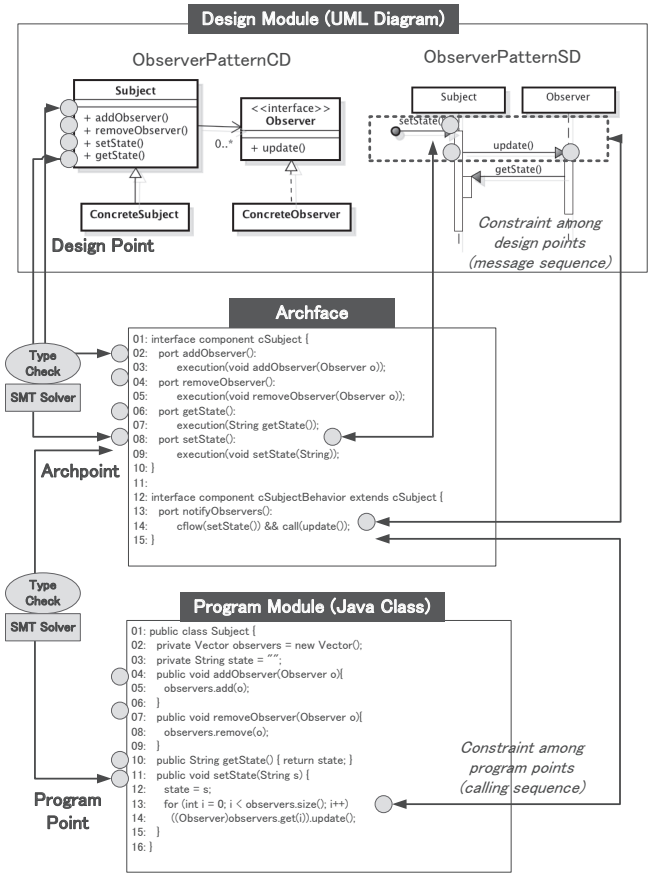

Figure 1. Archface-Centric MDD [12]

Archface, which supports *component-and-connector* architecture [1], consists of two kinds of interface, *component* and *connector*. The former exposes archpoints and the latter defines how to coordinate archpoints. A collaborative architecture can be encapsulated into a group of component and connector interfaces. Pointcut & advice in AspectJ is used as a mechanism for exposing archpoints (pointcut) and coordinating them (advice).

List 1 and 2 are component interfaces for the *Observer* pattern. *Archface* exposes archpoints from *ports*. Four port declarations (line 02-07) in List 1 correspond to the traditional interface in which each method declaration can be regarded as exposure of `method execution`. The `notifyObservers` port (line 12-13) exposes an `update call` archpoint that has to be called under the control flow of `setState`. The operator && is used to symbolize *Logical AND*. This archpoint is combined with an `update`

`execution` archpoint specified in a component interface for observers (List 2, line 02).

```
[List 1]
01: interface component cSubject {
02:  port addObserver():
03:       execution(void addObserver(Observer o));
04:  port removeObserver():
05:       execution(void removeObserver(Observer o));
06:  port getState(): execution(String getState());
07:  port setState(): execution(void setState(String));
08: }
09:
10: interface component cSubjectBehavior
11:           extends cSubject {
12:  port notifyObservers():
13:       cflow(setState()) && call(update());
14: }
```

```
[List 2]
01: interface component cObserver {
02:  port update(): execution(void update());
03: }
04:
05: interface component cObserverBehavior
06:           extends cObserver {
07:  port updateState():
08:       cflow(update()) && call(String getState());
09: }
```

List 3 is a connector interface specifying the coordination among archpoints exposed from component's ports. The execution of archpoints exposed from component interfaces is coordinated by `connects` (`multiple` indicates the connection is repeatable). In `notifyChange`, an `update call` archpoint in `cSubject` is bound to an `update execution` archpoint in `cObserver`.

```
[List 3]
01: interface connector cObserverPattern
02:                     (cSubject, cObserver);
03: interface connector cObserverPatternBehavior
04:           extends cObserverPattern {
05:  connects multiple notifyChange
06:    (cSubject.notifyObservers, cObserver.update);
07:  connects obtainNewState
08:    (cObserver.updateState, cSubject.getState);
09:  }
10: }
```

2.2 Abstraction-aware Traceability

The conformance to *Archface* can be checked by a type system that takes into account not only program but also design interfaces. The type checking is performed by verifying whether or not a design point (program point) corresponding to an archpoint exists in a design module (program module) while satisfying constraints among design points (program points) (e.g., the order of message sequences specified by `cflow`). The type checking can be implemented using an SMT (Satisfiability Modulo Theories) solver. The detail algorithm is shown in [11].

There is a bisimulation relation between a design and its code in terms of archpoints. The abstraction structure modeled in a design is preserved in its code. We can ignore program points that are not related to archpoints when we check the design traceability. That is, we cannot distinguish code from its associated design in terms of archpoints. The novel point of our approach is the realization of bisimulation in terms of a type system based on *Archface*.

2.3 Abstraction Refinement Process

Although the consistency between design and code in terms of an abstraction level specified by *Archface* can be preserved, it is a difficult problem to explore an appreciate abstraction structure and decide which abstraction level is reasonable.

In *AGAR* (Archface Guided Abstraction Refinement) inspired by CEGAR, an abstraction level is determined by archpoint selection. The level becomes high if the number of selected archpoints is few. On the other hand, an abstraction level becomes low if the number of selected archpoints is large. *AGAR* is a process of iteratively exploring appropriate abstraction between design and implementation. This process consists of three steps: 1) create an initial *Archface*, design, and code, 2) check the design traceability and measure an abstraction ratio, and 3) refine abstraction by modifying *Archface*, design, and code. We have to modify design or code when traceability is violated.

We propose *abstraction ratio*, a metric for measuring an abstraction level. The value of this metric is `1 - #ArchPoint / #ProgramPoint`. #ArchPoint is the number of architectural points in *Archface*. #ProgramPoint is the number of program points in code. We do not recommend "chasing metrics", because an adequate abstraction ratio changes corresponding to design situations. The determination of abstractions is a core task required to a developer. However, *if an abstraction level converges to a specific value in the process of AGAR, the value can be an appropriate abstraction level*. Form our experience, the structural aspect of the design tends to converge fast and the behavioral aspect tends to converge slowly.

We have to iteratively modify design models and code in the process of *AGAR*. This modification can be considered as refactoring not limited to an individual model and code but cross-cutting over them. We provide *MoveM2C* (Move from Model to Code) and *MoveC2M* (Move from Code to Model) as refactoring patterns to help abstraction refinement.

3. iArch

We are developing *iArch* for supporting *Archface*-Centric MDD. In this section, we show the overview of the *iArch* IDE.

3.1 Tool Overview

The *iArch* IDE shown in Figure 2 consists of the followings: 1) model & program editor, 2) *Archface* generator, 3) abstraction-aware compiler, and 4) abstraction metrics calculator. Using the model editor, we can make a design model and generate initial *Archface* descriptions. The abstraction-aware compiler checks whether or not an *Archface* is modeled by a design model and is implemented by the code.

The syntax of *Archface* is based on AspectJ pointcuts and is slightly complex for an ordinary developer. To relax this problem, *iArch* introduces syntactical sugar that can be translated into original *Archface* without losing the expressiveness. This syntactical sugar consists of the structural part and the behavioral part. The former is described as Java-like interface and the latter is specified using a notation similar to LTS (Labelled Transition System) as shown in List 4.

```
[List 4]
01: cSubject = (cSubject.setState->cObserver.update
02:             ->cSubject.getState->cSubject)
03: cObserver = (cObserver.update->cSubject.getState
04:             ->cObserver)
```

In Figure 2, an error is displayed because the sequence diagram does not conform to the *Archface*. If a developer considers that the *Archface* is correct and its abstraction level is appropriate, the

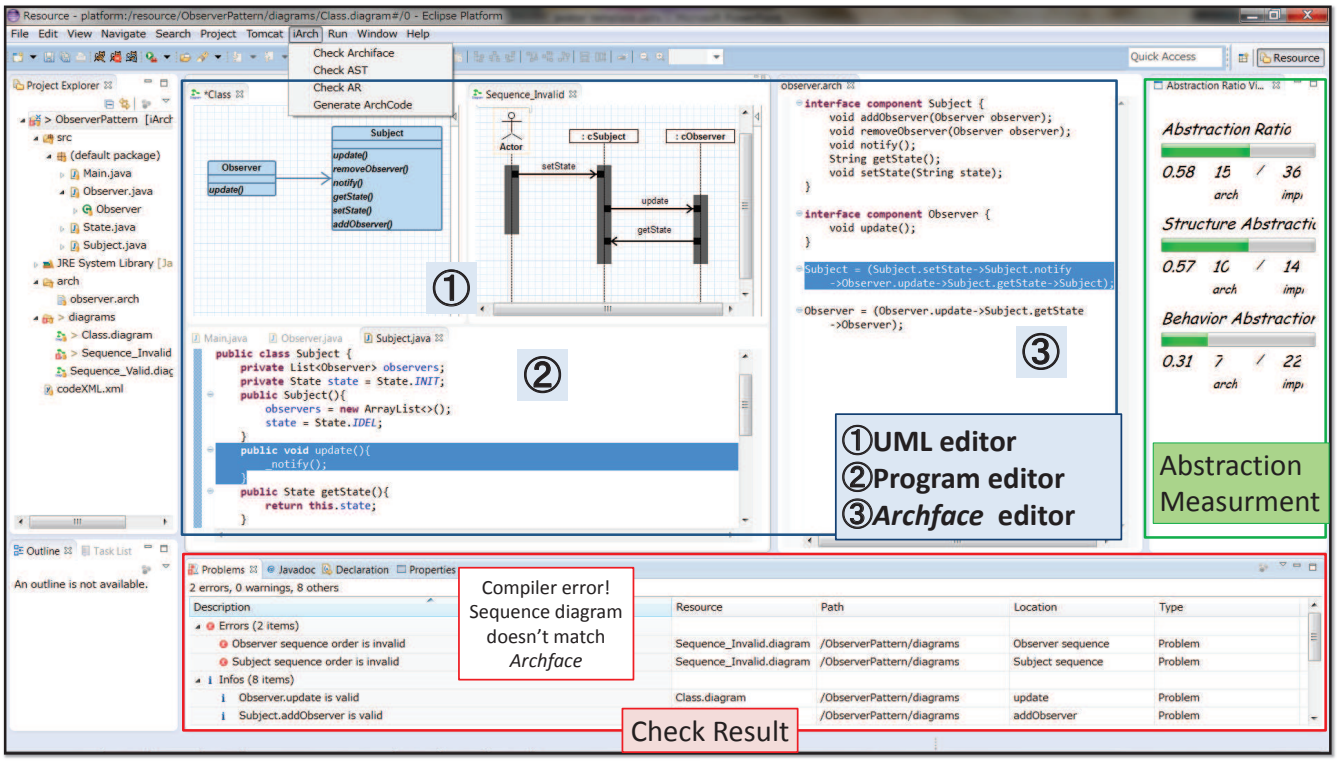

Figure 2. *iArch* IDE

developer has to change the design model. On the other hand, the abstraction level may be changed if the design model is reasonable. An error from the abstraction-aware compiler gives a developer an opportunity for obtaining an appropriate abstraction level.

3.2 Implementation

The *iArch* IDE is implemented as an Eclipse plug-in using EMF (Eclipse Modeling Framework) [5] and Graphiti (Graphical Tooling Infrastructure) [6]. The former is a tool that generates a model editor from a metamodel, and the latter provides a graphic framework for developing a graphical editor based on EMF. Currently, *iArch* supports class and sequential diagrams whose metamodels are basically the same as UML2 ecore metamodels.

4. Conclusion

The *iArch* IDE has the potential for exploring the next generation MDD, a type-based module integration to bridge design and code preserving an abstraction level. Our approach opens the door towards a new modularity vision whose concept is *"Not only program code but also a design model is a module."*

Acknowledgements

This research is being conducted as a part of the Grant-in-aid for Scientific Research (B) 23300010 and Challenging Exploratory Research 25540025 by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

[1] Allen, R. and Garlan, D.: Formalizing Architectural Connection, In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*, pp.71-80, 1994.

[2] Bass, L., Clements, P., and Kazman, R.: *Software Architecture in Practice (2nd edition)*, Addison-Wesley, 2003.

[3] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M.: *Pattern-Oriented Software Architecture –A System of Patterns*, John Wiley & Sons, 1996.

[4] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H.: Counterexample-Guided Abstraction Refinement for Symbolic Model Checking, In *Journal of the ACM*, Vol. 50, Issue 5, pp.752-794, 2003.

[5] EMF, http://www.eclipse.org/modeling/emf/, 2014.

[6] Graphiti, http://www.eclipse.org/graphiti/, 2014.

[7] Kiczales, G., Hilsdale, E., Hugunin, J., et al.: An Overview of AspectJ, In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2001)*, pp.327-353, 2001.

[8] Kramer, J.: Is Abstraction the Key to Computing? Communications of the ACM, Vol. 50 Issue 4, pp.36-42, 2007.

[9] Taylor, R. N. and Hoek, A.: 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, 2007.

[10] Ubayashi, N., Nomura, J., and Tamai, T.: Archface: A Contract Place Where Architectural Design and Code Meet Together, In *Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010)*, pp.75-84, 2010.

[11] Ubayashi, N. and Kamei, Y.: Verifiable Architectural Interface for Supporting Model-Driven Development with Adequate Abstraction Level, In *Proceedings of the 4th International Workshop on Modelling in Software Engineering (MiSE 2012) (Workshop at ICSE 2012)*, pp.15-21, 2012.

[12] Ubayashi, N. and Kamei, Y.: Design Module: A Modularity Vision Beyond Code, In *Proceedings of the 5th International Workshop on Modelling in Software Engineering (MiSE 2013, Workshop at ICSE 2013)*, pp.44-50, 2013.