iArch - An IDE for Supporting Abstraction-aware Design Traceability

Di Ai, Naoyasu Ubayashi, Peiyuan Li, Shintaro Hosoai and Yasutaka Kamei

Kyushu University, Fukuoka, Japan

Keywords: Architecture, Interface, Abstraction, Traceability, Type System, Model-Driven Development.

Abstract: Abstraction has been an important issue in software engineering. However, it is not easy to design an architecture reflecting the intention of developers and implement the result of a design as a program while preserving an adequate abstraction level. To deal with this problem, we provide iArch, an IDE (Integrated Development Environment) for supporting abstraction-aware traceability between design and code. The iArch IDE is based on Archface, an architectural interface mechanism exposing a set of architectural points that should be shared between design and code. An abstraction level is determined by selecting architectural points.

1 INTRODUCTION

Abstraction has been an important issue in software engineering research (Kramer, 2007). MDD (Model-Driven Development) is a promising approach to deal with abstraction. An application can be developed at a high abstraction level by using a DSL (Domain-Specific Language) or a DSML (Domain-Specific Modeling Language). We do not need program code, because it can be fully generated from its design model if necessary. However, programming has not yet disappeared from most software development projects, because the approach of DSLs can be only applied to mature application domains. In ordinary development, both design activities and programming have their own roles. Although separation of design and implementation concerns is important, it is not easy to design an architecture reflecting the intention of developers and implement the result of a design as a program while preserving architectural correctness and an adequate abstraction level.

To deal with this problem, we previously proposed an approach that treats a design model as a first-class software module (Ubayashi and Kamei, 2013). A design model such as a UML diagram is regarded as a design module. To realize design modules, we introduced Archface (Ubayashi et al., 2010), an architectural interface mechanism. Archface exposes a set of architectural points that should be shared between design and code. Archface plays a role as a design interface for a design module and as a program interface for a program module. As a well-modular program can be obtained by defining deeply considered program interfaces between program modules, an excellent separation of concerns among design and program modules can be captured by defining well-balanced Archface. The traceability between design and code can be automatically maintained by the type system of Archface. Design and code are synchronized when both a design model and its code conform to the same Archface. Abstraction is taken into account in this synchronization, because an abstraction level is determined by selecting shared architectural points declared in Archface.

This paper provides iArch, an IDE (Integrated Development Environment) for supporting abstraction-aware traceability between design and code. The iArch IDE consists of the followings: 1) model & program editor, 2) Archface generator, 3) abstraction-aware compiler, 4) abstraction metrics calculation, and 5) refactoring support. The contribution of this paper is to show how to realize an Archface-Centric MDD tool based on a new type system that can be applied to not only a program but also a design model.

This paper is structured as follows. Archface-Centric MDD is introduced in Section 2. The implementation of iArch is illustrated in Section 3. Discussion and future work are provided in Section 4.

2 ARCHFACE-CENTRIC MDD

In this section, we provide a background on Archface-Centric MDD, a foundation of iArch, by excerpting from our preliminary work (Ubayashi and Kamei, 2013).
2.1 Basic Concept

Figure 1 shows the relation among design modules, program modules, and Archface, an interface for bridging them. 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. A design specification consists of multiple design modules in which each module represents an individual design concern. Design modules support MDSOC (Multi-Dimensional Separation Of Concerns) (Tarr et al., 1999). 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 (Kiczales et al., 2001). We focus on archpoints embedded in class and sequence diagrams, because structural and behavioral aspects of software architecture (Bass et al., 2003) 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.

2.2 Archface, Design and Code

We illustrate design and program modules using the Observer pattern as an example. The Observer pattern consists of a subject and observers. When the state of a subject is changed, the subject notifies all observers of this new state. Figure 2 shows the relation between these modules and Archface.

2.2.1 Archface

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 archpoints. Hierarchical definitions are possible, because both interfaces support inheritance. A collaborative architecture can be encapsulated into a group of component and connector interfaces. Pointcut & advice in AspectJ is used as a mech-
anism for exposing archpoints (pointcut) and coordinating them (advice).

List 1 is a component interface for a subject. Archface exposes archpoints from ports. Four port declarations (line 02-07) correspond to the traditional interface in which each method declaration can be regarded as exposure of method execution. The notifyObservers port (line 11-12) exposes an update call archpoint that has to be called under the control flow of getState. 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 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 4 and 5 are Java classes implementing the Archface.

2.3 Abstraction-aware Traceability

To integrate design and program modules, each design module models its Archface and each program module implements the same Archface. The conformity 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 $cflow$). Although traditional types are structural—sets of method signatures, Archface is based on archpoints including behavior—specified by the order of archpoints because a design model imposes structural or behavioral architectural constraints on a program.

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. Bisimulation is an important concept in the research field of process algebra (Milner, 1989). Two process are bisimilar if each process cannot be distinguished from the other. A sequence of archpoints can be regarded as a process if we regard the sequence as an LTS (Labeled Transition System). 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.

2.2.2 Design and Program Modules

Both design and program modules are the same as traditional UML diagrams and code. However, there is a crucial difference. An interface, Archface, resides between them and it makes them software modules. ObserverPatternCD is a class diagram, and ObserverPatternSD, a sequence diagram shown in Figure 2 are design modules faithful to the Archface declared in List 1, 2, and 3. A program module is also the same as a traditional module such as a Java class. List 4 and 5 are Java classes implementing the Archface.
3 iArch

3.1 Tool Overview

The iArch IDE consists of the followings: 1) model & program editor, 2) Archface generator, 3) abstraction-aware compiler, 4) abstraction metrics calculation, and 5) refactoring support. 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. If the code does not exist, the compiler generates the code conforming to the design. The compiler supports the preservation of consistency between design and code in terms of an abstraction level specified by Archface. However, it is an open question to explore an appropriate abstraction structure and decide which abstraction level is reasonable. We provide abstraction ratio, a metric for measuring an abstraction level, and refactoring patterns for abstraction refinement. The abstraction ratio is calculated as

\[ \text{abstraction ratio} = \frac{\text{number of arch points}}{\text{number of program points}}. \]

This ratio helps a developer to determine which arch point should be available in a design. We have to iteratively modify design models and code until we can capture an adequate abstraction level. 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. The MoveM2C pattern 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. The MoveC2M pattern moves a code concern to a design concern. This pattern, the reverse of the MoveM2C pattern, is applied to the situation in which a developer 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.

Figure 3 is a snapshot of iArch. Currently, model editor and type checker are provided. The left side of Figure 3 is a sequence diagram of the Observer pattern and the right side is an Archface definition.

3.2 Syntactical Sugar for Archface

The syntax of Archface is based on AspectJ pointcuts and is slightly complex for an ordinary developer. However, there are several merits to introducing AOP notation to an interface mechanism. For example, an Archface definition can be automatically translated...
into the AspectJ code that checks the behavioral aspect of a design model at runtime. Although the type system in the abstraction-aware compiler can check the design traceability using static analysis, it cannot cover all the behavioral aspects. Some design properties have to be checked by runtime testing.

To relax the complexity of Archface descriptions, the iArch IDE 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 LTS-like notations. Currently, only event-based message sequence can be specified in the latter case. The support of other aspects such as data flow specifications is future work. List 6 is the behavioral specification using the LTS-like syntactical sugar.

```
<table>
<thead>
<tr>
<th>List 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>01: cSubject = (cSubject.setState-&gt;cObserver.update)</td>
</tr>
<tr>
<td>02: -&gt;cObserver.getState-&gt;cObserver</td>
</tr>
<tr>
<td>03: cObserver = (cObserver.update-&gt;cSubject.getState)</td>
</tr>
<tr>
<td>04: -&gt;cSubject.getState-&gt;cSubject</td>
</tr>
</tbody>
</table>
```

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. List 6 corresponds to notifyObservers (List 1, line 11-12), updateState (List 2, line 06-07), and List 3. The Observer pattern can be represented as cSubject || cObserver (|| is an operator for process composition). Introducing the LTS-like notation, verification tools such as LTSA (LTS Analyser) (Magee and Kramer, 2006) can be applied to check a bisimulation relation between design and code.

3.3 Traceability Check in iArch

In Figure 3, an error is displayed because the sequence diagram does not conform to the Archface. This Archface specifies that the update method of the Observer class should be called after the notify method of the Subject class is called. However, the update method is directly called in the sequence diagram. If a developer considers that a design model is correct, the Archface descriptions have to be modified as List 6. In this case, the abstraction level becomes higher than that of Figure 3. We do not have to consider whether or not notify is called in the code. That is, there is a freedom in the program implementation. If a developer considers that the Archface is correct and its abstraction level is adequate, the developer has to change the design model. As mentioned here, an error from the abstraction-aware compiler gives a developer an opportunity for obtaining an appropriate abstraction level.

Our approach can be applied to check the consistency among diagrams. For example, a method declaration in a class diagram is detected as an error if the corresponding message receive does not exist in a sequence diagram. If a developer forgets to remove the notify method from cSubject when modifying to List 6, the abstraction-aware compiler generates an error. This can be verified by checking the inconsistency within the Archface definition. This error detection is available only if the associated archpoints are declared in the Archface. We can ignore other inconsistencies that a developer does not need to concern themselves with.

3.4 Implementation

The iArch IDE is implemented as an Eclipse plugin using EMF (Eclipse Modeling Framework) (EMF, 2013) and Graphiti (Graphical Tooling Infrastructure) (Graphiti, 2013). 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. We added only one model element corresponding to archpoints to the ecore metamodels. A design model conforms to the UML2 standard. Since Archface is a kind of DSL for interface descriptions, we implemented Archface using Xtext (Xtext, 2013), a framework for developing text-based DSLs.

4 CONCLUSIONS

As claimed in Section 1, abstraction plays a key role in software design. We show related work concerning abstraction and traceability check. Cassou, D. et al. explored the design space between abstract and concrete component interactions (Cassou et al., 2011). They provided an ADL (Architectural Description Language) for Sense/Compute/Control applications, and described compilation and verification strategies. Our approach provides a general model for design traceability, because the model can be applied to not only Sense/Compute/Control applications but also other systems. As one of the important research directions in the field of software design, Taylor et al. pointed out the need for adequate support for fluidly moving between design and coding tasks (Taylor and Hoek, 2007). To deal with this problem, Y. Zheng and R. N. Taylor proposed 1.x-way architecture-implementation mapping (Zheng and Taylor, 2012) for deep separation of gen-
generated and non-generated code. We can correspond the former and the latter to design concerns and implementation concerns, respectively. However, it is not easy to change the separation of concerns between generated and non-generated code. Aldrich, J. et al. proposed ArchJava (Aldrich et al., 2002) that unifies architecture and implementation, ensuring that the implementation conforms to architectural constraints. ArchJava does not contain such an idea that a UML model can be regarded as a module. We think that it is important to improve MDD using ordinary UML and programming languages because they are familiar to many developers in industry. Steel, J. et al. introduced model types (Steel and Jézéquel, 2005) that treat models as a collection of interconnected objects and deal with the relationships defined in MOF (Meta-Object Facility). In our approach, design and program modules implementing the same archpoints belong to the same type.

This paper introduced iArch, an IDE for Archface-Centric MDD. Although current Arch is a prototype, it has the potential for exploring the next generation MDD: Current MDD takes a transformation approach such as QVT (Queries/Views/Transformations) or ATL (ATLAS Transformation Language) to generate code from a design model. On the other hand, our approach is a type-based module integration to bridge design and code preserving an abstraction level. As a next step, we plan to enhance Archface to support not only abstraction but also uncertainty. In most software development, models tend to contain uncertainty, because all of design concerns cannot be captured at the early phase. Uncertain concerns, which crosscut over design models and code, can be specified by Archface based on AOP. MDD embracing uncertainty is one of the important research topics.

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


Ubayashi, N. and Kamei, Y. (2013). Design module: A modularity vision beyond code—not only program code but also a design model is a module—. In Proceedings of the 5th International Workshop on Modelling in Software Engineering (MiSE 2013) (Workshop at ICSE 2013), pp.44-50. IEEE.

