NANZAN-TR-2012-02

Aspect-Oriented Software Architecture for CDI Tools: toward PLSE Construction

Masami Noro and Atsushi Sawada

June 2012

Technical Report of the Nanzan Academic Society
Information Sciences and Engineering

Aspect-Oriented Software Architecture for CDI Tools: toward PLSE Construction

Masami Noro
yoshie@se.nanzan-u.ac.jp

Atsushi Sawada
sawada@se.nanzan-u.ac.jp

Department of Software Engineering, Nanzan University

June 16, 2012

Abstract

This paper describes the aspect-oriented software architecture for a code inspection tool. Through the development of a production quality code inspection tool, crosscutting concerns: - internal data model, - language processing, - data traverse, - inspection logic, and - decoupling control are identified. With - Composite, - Interpreter, - Visitor, - Mediator, - Command and - State patterns, the software architecture is implemented. The usefulness of the architecture is discussed. The state transition model for the specification to the inspection logic is introduced and its practicality is demonstrated with several examples. Practice of PLSE with the software architecture is also discussed and we concluded that we have already prepared most of core assets we need for the practice.

1 Introduction

Code inspection is just a code-level but promising approach to detect possible faults in software. Tools for code inspection[8] for various programming languages have been developed and widely used for real projects. Most of the tools are useful for detecting the faults but the functionalities of the tools are built-in and are never customizable. On the other hand, requirements to the functionalities of the inspection varies from a project to another. In this sense, tailoring the functionality is one of the most important non-functional requirement to code inspection tools.

We achieved to realize a basis for customizable code inspection tools for Java. There have been two open questions to design and implement a customizable code inspection tool. One is what is a suitable structure of the tool for the function customization and another is how you write requirements for the inspection. We responds to the questions in the way that we give easy-to-modify structure of inspection tools for developers. The structure also facilitates user customization of the tools.

The first question concerns about software architecture. We constructed a software architecture for modifiable application. It is based on a software architecture for a language processor. We designed the architecture which can tell frozen spots from hot spots in an application framework.

The second question is on specification. We need an understandable and easy-to-write specification method. It is also required to be suitable for machine manipulation. We adopted a state transition model to meet these needs. The model is towards automatic generation and formal manipulation.

FindBugs[4], PMD[12], CheckStyle[11], JLint[1], and so on are practical code inspection tools but are not customizable at all. Only CX-Checker[7] provides customization facility. The facility it gives is still hard to use for the users. Our Java code inspection tool is for much more flexible suite for use customization. METAL[3] is another alternative for the customization. They provide their specialized language METAL for writing specifications to code inspection. It is, however, just a script language and they provide it as a language in an isolated environment. We took the idea of METAL and generalized it to work on an aspect-oriented software architecture. The architecture is the open-ended solution to open questions to code inspection tools

Thus, the inherent problem to be solved, however, when a customizable code inspection tool is considered is if we could define

1. architecture to separate parsing concern from inspection logic,
2. reusable components for building the inspection logic, and
3. specification method for the inspection logic.

2 Software Architecture

This section gives software architecture solution to the problems mentioned in Introduction.

2.1 Requirements for Code Inspection Tool

We have investigated typical code inspection tools to find out requirements. Non-functional requirements are also researched when we develop, with a software house, a production quality code inspection tool for Java called JCI[1] [9]. In general, functional requirements to code inspection can be categorized into ones on parsing, control flow, or data flow. In addition, we recognized the following nonfunctional requirements[5] when we had developed JCI.

- Performance efficiency (both time and resource)
- Usability
  - Customizability
- Maintainability
  - Modifiability
  - Flexibility

This paper concerns about architecture. Thus, we select Customizability, modifiability, and flexibility that relate to architecture design.

2.2 Traditional Approach to Development of Code Inspection Tool

There had been no software architecture for code inspection tools but crafting techniques. That is, a code inspection tool has been conceived as a better compiler that has richer functionality than the compiler has. Nowadays, syntax-directed or attribute grammar approach to automatic generation is widely used for building a compiler as well as a compiler-like software tool such as a code inspection tool. Application logic is attached to a syntax rule when they use the approaches. A symbol table and an intermediate code which is mostly an abstract

[1] JCI is a registered trademark of ATM Japan, Ltd.

syntax tree are defined as reusable components. That is, operations to a symbol table and an intermediate code are standardized to be a library for building a compiler. In addition, syntax-directed approach can be conceived as a way to generating an application framework for a specific programming language. The traditional approach to building code inspection tools here we mention, however, does not solve none of the problems above.

2.3 Architecture Design

The first software architecture for a compiler was given as an example to present what is software architecture by Shaw and Garlan[10]. The architecture is in a natural design and can be widely accepted. However, no separation of concern was considered. We, as mentioned in the previous section, need to separate a parsing concern from an inspection logic concern in order to establish the basis for code inspection tool customization. Moreover, we need to make an aspect-oriented software architecture towards software construction with reusable components.

Without any doubt, object-oriented is a primary concern. That is, we design a code inspection tool in an object-oriented fashion. We recognize five concerns crosscutting across one another and the object-oriented concern:

- internal data model,
- language processing,
- data traverse,
- inspection logic, and
- decoupling (this concern is for separating the traverse and the internal data model access from the inspection logic).

The internal data model concern is about internal description of a program inspected. It is on a symbol table, an abstract-syntax tree, a control-flows graph, and data-flows graph. The language processing concern is mostly related to front-end processing, that is, scanning, parsing, and programming language semantics check. The data traverse concern is on pulling relations among syntactical nodes. The inspection logic concern defines decomposition of inspection logic. It is mostly on functional decomposition of the functionality a code inspection tool provides. The decoupling concern is for implementing concurrency between a traverse component and an inspection component. It is also for internal data access wrapping.

Our primary separation of concern is on that of an internal data model and an inspection logic. We need the separation since modification to the inspection logic varies a lot from a requirement to code inspection to another while changes to internal data model is less. Since the code inspection task is a set of activities for accessing names in a symbol table and a syntax tree, a control-flows or a data-flows graph in an intermediate code, codes for internal data model access crosscuts all other application codes. From the reason that we would like to take state transition model to specify an inspection logic, the inspection logic must be designed as an event based subsystem. Components decomposed in the traverse concern trigger events such as accessing a data declaration node, reaching a specific statement node, and so on. Information access from the inspection logic into internal data model must be wrapped by standardized access messages since we need to retain internal data model independent from how it is used. To realize these two decoupling mechanisms defined by the decoupling concern, we introduced two components: the access logic and the message service provider.

2.3.1 Static Structure

As in figure 1, ten components are in the architecture. They represents a family of components which have same functionality, and/or a set of components to fulfill its functionality. The front-end processor consisting of the scanner, the parser, and the semantic checker reads a source program and constructs internal database of the program. The internal data model is a set of the symbol table, the abstract syntax tree, and the control-flows and data-flows graph cutting across the tree. The inspection logic is one of the most important components in the system. It implements code inspection functionalities. It reads data from the internal database through the access logic. It traverses the abstract syntax tree, the control-flows graph, and data-flows graph through the message service provider. The access logic and the provider are for decoupling data access from behavior of the inspection logic.

Each identified crosscutting concern above corresponds to an aspect in the figure 1. The language processing aspect is for syntax and semantics checking. For the sake of language independence, we introduced the aspect. It relates the

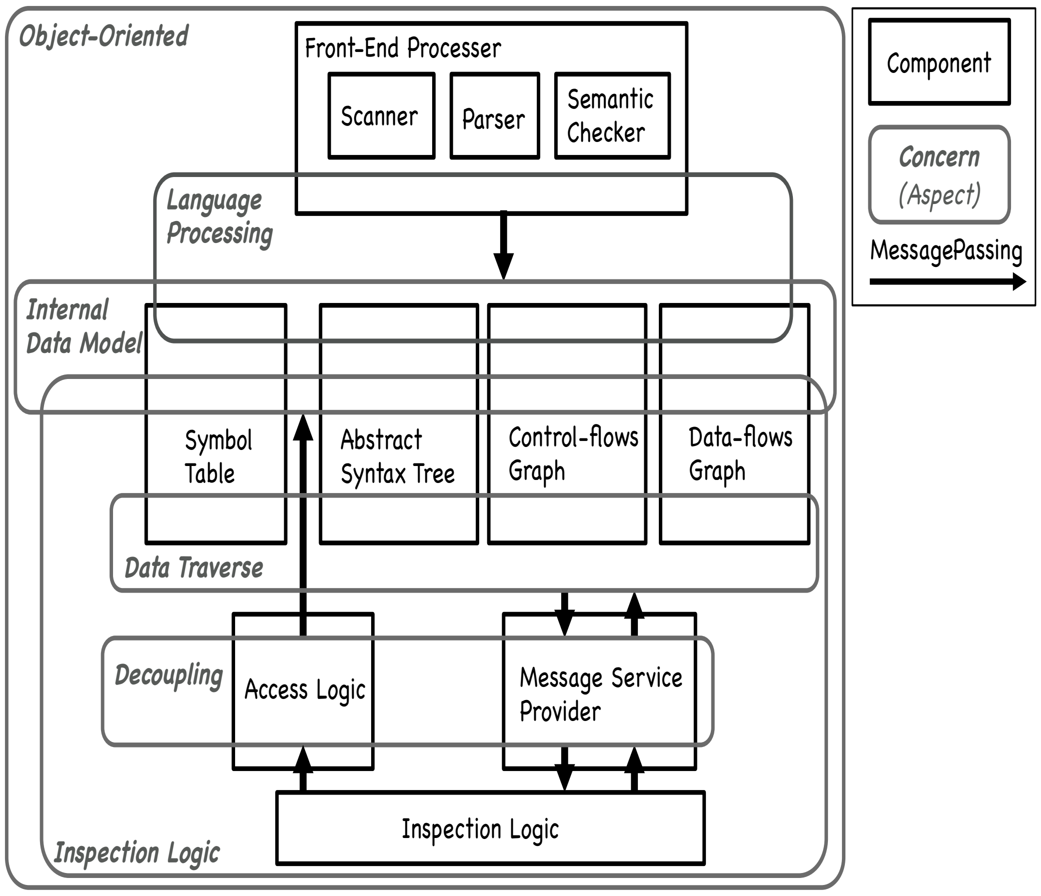


Figure 1: Aspect-Oriented Architecture for Code Inspection Tool

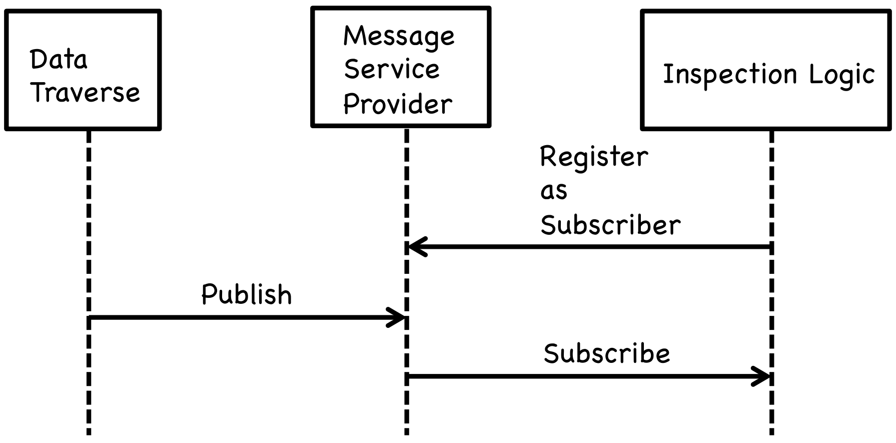


Figure 2: Dynamic Behavior of Architecture

internal data model aspect. We try to separate internal data model from parsing facilities. The decoupling aspect separates the internal data model from the inspection logic. To design the inspection logic component in an event-based architecture gives two advantages to code inspection tool design: independency and specification. The independency means that the inspection logic and the internal data model can be modified with less effects on one another. We mean by the specification that a designer or implementer can specify the functionality of an inspection as a sequence of events. That is, she can writes the specification in a state transition model. For the cooperation of the data traverse with the code inspection, we use publish-and subscribe architecture to decouple these two components. The publisher is one of the abstract syntax tree, the control-flows graph, or the data-flows graph. The subscriber is the inspection logic. The access logic component wraps access to the internal database from the inspection logic. It is for the independence of the internal data model.

2.3.2 Dynamic Behavior

In general, there are two separated procedure in a code inspection tool: the front-end and the back-end. The front-end is on the syntax and semantic checking, and creating internal database of a program. The back-end is for code inspection. With the reason we mentioned above, we separate the traverse function from the inspection function. The dynamic behavior of these two functions is cartooned in figure 2 As in the figure, the message service provider mediates the traverse logic and the inspection logic. A concrete inspection logic component registers topics it interests and waits the events relating the topic. The message service provider give the subscription when it receives publish message from the traverse logic.

3 Architecture Implementation with Pattens

For the purpose of generality, we basically adopted design patterns[2] to implement our software architecture. That is, we use the patterns since we would not like to write software in a specific aspect-oriented programming language such as AspectJ, etc. To implement the aspects in figure 1, we borrow Composite, Interpreter, Visitor, Mediator, Command, and State patterns. This section describes that how and why the patterns are used.

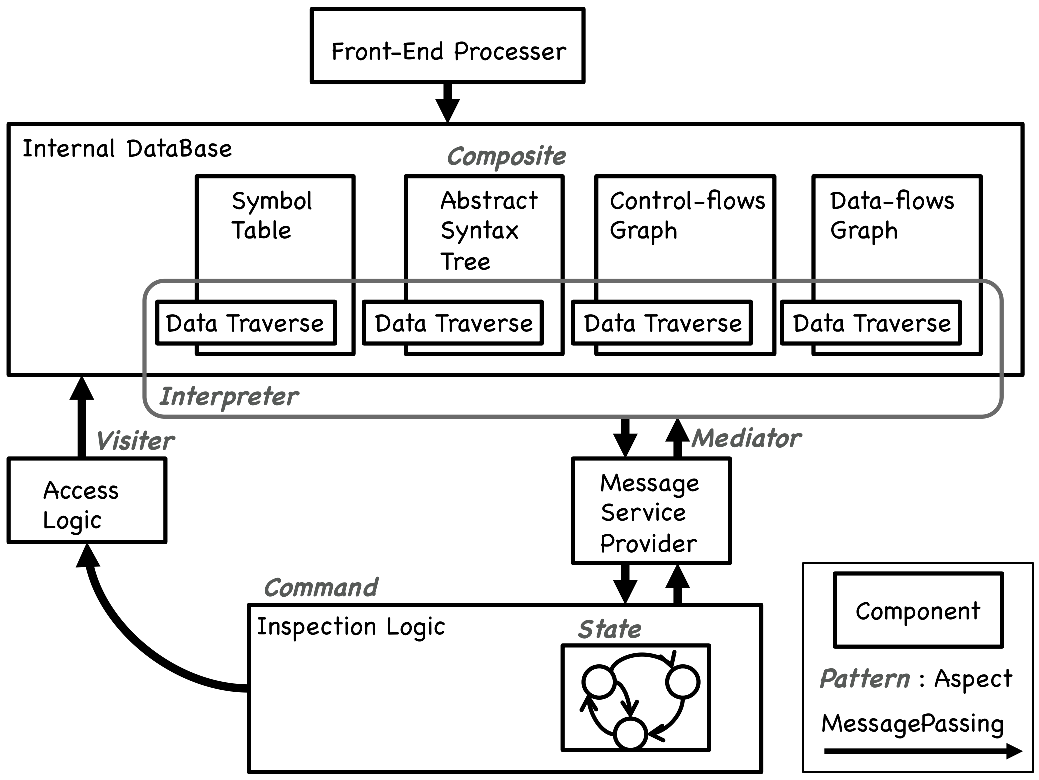


Figure 3: Implementation of Architecture with Patterns

Figure 3 shows a natural implementation of our architecture in figure 1. As in the figure, we use Composite pattern for representing main data structure of the internal database as an object model. Interpreter pattern is used to implement the traverse aspect, actually, the event issuer which traverses a tree or a graph and raise an event every time it visits the node. Visitor is for the implementation of database access logic and we make use of Mediator for the message service provider which realizes the publish and subscribe architecture. Both patters are for the implementation of the decoupling aspect. The inspection logic is packed in a Command object and State pattern represents state transition specification of the logic.

3.1 Composite for Internal Data Model

Composite is a pattern for presenting tree structure. The main data structure we manipulate in a code inspection tool is, without any doubt, the tree whose syntax is defined by the programming language (Java in our case). That is, there was no other selection than adopting Composite pattern for the abstract syntax tree in our internal data model. A Control-flows or data-flows graph is implemented as a directed acyclic graph. A node of the graph is one in the abstract syntax tree and links cut across the tree, then forms the graph. The symbol table is realized as a table of links which relates a name reference node to its declaration node. The link also cuts across the abstract syntax tree.

Summing up the discussion above, we had no other alternatives than selecting Composite for the abstract syntax tree. We took the design that the graphs and the table are realized as a set of instance variables which reference other nodes in a node object. Thus, we took the natural model in a straightforward way.

3.2 Interpreter for Traverse

We select Interpreter pattern for the implementation of the traverse aspect not Observer pattern. Since the traverse component walks around the abstract syntax tree or a graph in the tree whose syntax is predefined, Interpreter is the best fit for manipulating the tree. That is, we did not choose Observer because it puts more emphasis on its mechanism for messages passing than the interpreter. Moreover, Observer pays less attention to the syntax it observes. Thus, we choose Interpreter because the syntax is predefined and its traverse is deeply related to the syntax.

3.3 Visitor for Database Access Logic

The database access logic aspect is implemented with Visitor pattern. Since we would like to separate data definition from its access policy, Visitor is the most natural alternative we took. As the implementation of our tools proceeds over and over, the access logic to internal database are standardized more and more. In the current version, it is well defined enough to fix the logic. We tried to use Strategy pattern but we did not since the access logic is already well-defined for our application.

3.4 Mediator for Message Service Provider

To implement the message service provider component, we choose Mediator pattern. The main reason why we select Mediator pattern is that the message passing would occur bi-way. In detail, the message service provider holds publish-and-subscribe registry and the traverse component is usually a publisher. Since several inspection logics are executed at the same time, the inspection logics wait events on multiple topics. On the other hand, the inspection logic component may be a publisher when it triggers the traverse. The traverse request is one of to the abstract syntax tree, the control-flows graph, or the data-flows graph. That is, traverse side components may subscribe on multiple topics. Thus, each could be a publisher and then we employed bi-way mediator to implement the message service provider. Instead, Proxy or Decorator pattern may be used as a wrapper to access an inspection logic component from a traverse component, or vice versa. The patterns are for one-way use. They may lead to more complicated structure than the case in which Mediator is used.

3.5 Command and State for Inspection Logic

Command pattern packs and parameterizes a procedure in it. We used to implement a inspection logic as a collection of Visitors. At that time, we did not introduce a traverse component. Facade pattern was used instead to group Visitors realizing the logic. That is, the logic is not separated from the dat model enough to be the command.

Inside a command component, a state transition machine is implemented with State pattern. We choose the pattern because it gives the most natural object-oriented structure for the implementation of state transition specification.

4 Specification to Code Inspection Tool

This section gives the state transition mode we use for write a specification to an inspection logic. Several examples of specification are presented to demonstrate the practicality of the model.

4.1 State Transition Model for Specification

A problem to be solved by code inspection is, in nature, one on definition and usage of a value of a variable (we call the problem a def-use problem hereafter). As mentioned earlier, METAL[3] is a language which is designed to model def-use problem as a state transition. That is, an event to state transition is raised when a specific node is visited during the traverse in a control-flows graph, or a data-flows graph. The following is a typical solution to a def-use problem.

1 When a node for value definition is visited, the fact was memorized in an inspection logic component.

2 An event visiting a node for value usage triggers the check if the value was defined or not.

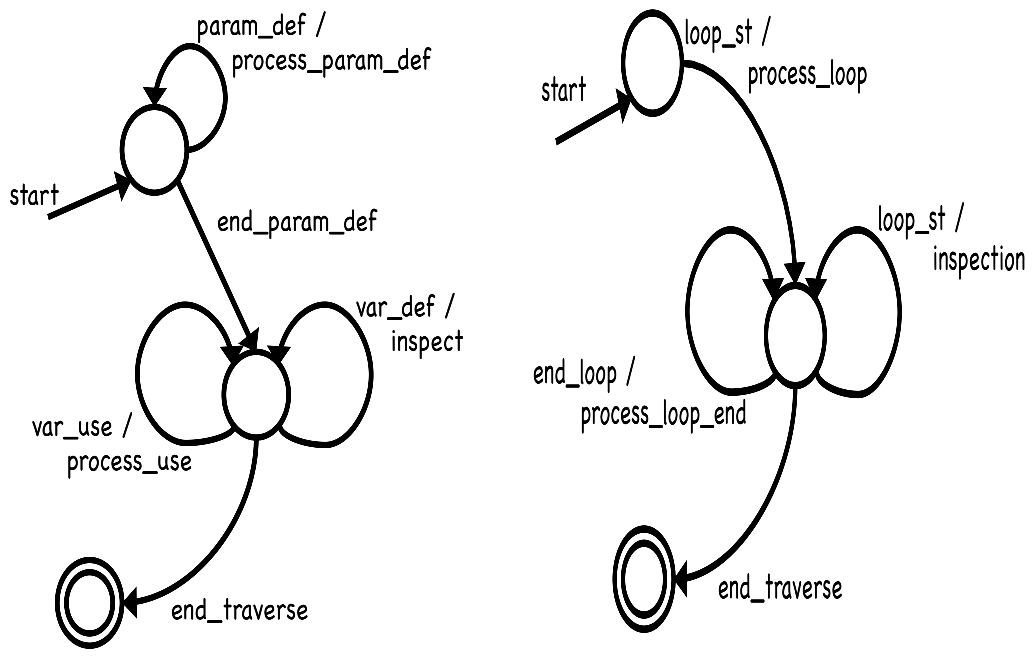
Thus, in the case of handling a flow graph, the state transition model works. Moreover, a state transition diagram such as the state machine diagram in UML can be used to write the specification in this approach. It is important for a developer if there is a well-accepted notation. Therefore, we employed the state transition model for the specification of code inspection.

Things are little different when we try to apply the state transition model to a parsing related check including semantics. In the formal language theory, the check can be performed with a push-down automaton since the class of the grammar manipulated is context-free. A finite-state automaton, that is, a state transition model is for a regular grammar class problem. There must be some cheat to write a specification for a parsing related problem. Instead of a proper manipulation of a stack as a push-down automaton does, we just count depth of a nest we handles. For example, we just count up the nest on control structure when we reach every if-statement node if we would like complain with too deep nest of if-statement. Since we could assume that parsing including semantic check is completed in a front-end process, we could always almost avoid troublesome stack related manipulation. The way we implemented an inspection to a parsing problem is as follows:

1 A traverse component visits abstract syntax tree node in a pre-order and raises an event when the node is newly visited.

2 An event visiting a target node triggers the inspection action, mostly just count up.

3 We complain when the erroneous condition is satisfied, e.g. too deep nest, usage does not mach declaration, and so on.



(a)Specification to Data-flow problem (b)Specification to parsing problem

Figure 4: Examples of State Transition Specification

4.2 Specification Example

Figure 4 is examples of state transition specifications to inspection logics. Figure 4 (a) shows the case of overwriting value of a formal parameter before its use. (b) shows the case of the same control variable in a nested loop.

Here is source code complained with the logic whose specification is in figure 4 (a).

```
public void m(String s)  {
  s = null; // warning here
  if (s.equals("")) ...
}
```

We inspect if there is a value definition statement for a formal argument before the value via a message is used. This can be inspected by traversing data-flow graph for formal parameters of the program. The `param_def` event in figure 4 (a) trrigers the `process_param_def` action. In the action, the name of the formal parameter is memorized and construction of data-flows graphs for the parameters are requested. The `var_use` event causes the action `process_use` in which we check if the corresponding parameter is used. In the event of `var_def`, the inspection is performed where we check if it is a non used parameter.

The inspection if the nested loops have the same control variable can be specified as in figure 4 (b). A program having the following fragment will be complained.

```
for (i = 0; i < N; ++i) {
   ...
   while (i < M)  { ... }
}
```

This check can be performed by traversing the abstract syntax tree. The `process_loop` action triggered by `loop_st` event which denotes that one of a loop statement node is visited sets up the inner loop check. Succeeding `loop_st` events causes the inspection action. In the action, we may check if there is any variable with the same name in the condition of the loop visited before.

5 Discussion

We have implemented, with software company, JCI which is a production quality code inspection tool for Java in our previous architecture. The first version of the tool has thirty-six requirements for the inspection. In the previous architecture, decoupling inspection logic from data model manipulation was not clear. It took about two years for us to complete the first development. As times passes, the requirements are added and we developed new codes while maintaining existing code. It was lasting two years, then at last, we made up our mind to do architecture-level refactoring. In this moment, each requirements has become complicated more and more although the number of the requirements stay the same. We are now rewriting all of the code except for front-end components under our architecture described in this paper. All specifications to inspection logic are written in state transition model.

5.1 Usefulness of Software Architecture

Objectives to constructing our software architecture is customization and specification. The final customization is performed by a user who develop codes with our code inspection tool. Unfortunately, the software architecture itself cannot achieve the goal. However, it is the inherent role of software architecture that it establishes the basis. In this sense, we have prepared the basis towards the final customization. First, through the development of JCI, we have defined reusable components for both data model and its access logic. We have also standardized events and actions in an inspection logic. Thus, we believe our software architecture established the basis for the final customization.

From the code view, our architecture defines an application framework of a code inspection tool. Codes are almost standardized and now we can tell what parts are frozen spots and what for hot. One of typical hot spots is an action of a state transition machine. We have already define most of reusable components which fill out the blanks, i.e. hot spots, for actions. The only missing piece is the generator from logic specification to Java code. With the generator, we would almost approach to the final customization.

5.2 Practicality of State Transition Specification

As mentioned above, user-friendly specification method is indispensable to the final customization. We took a state transition model for the specification. We have already specified all of current requirements to JCI with the approach. The state transition model is also design specification. In the development of the current version of JCI, we could write all inspection logic code from the state transition model. That means the approach is practical and works well for writing specifications to the inspection logic. These means our approach to the specification is practical enough to develop a production quality tool.

5.3 Towards PLSE

The most difficult problems in a practice of PLSE[6] is how we can map a requirement to a product to components in the product. Fortunately, one-to-one mapping from a requirement to a corresponding inspection logic or vice versa is ensured. Moreover, we have already constructed our software architecture described in this paper. The architecture defines an application framework and we realized that the main hot spot is the inspection logic. We also realized that state transition model for the specification to the inspection logic works well. This discussion shows that we almost are to stand the PLSE practice of a code inspection tool development.

The core assets in this PLSE practice consists of our software architecture, reusable components we have defines, specifications and specification method we propose, i.e. state transition model, and software tools, i.e. generators.

In the domain engineering process of this PLSE practice is very straightforward. All a user has to do is just selecting the requirements she needs. Since one-to-one mapping from the requirement to the inspection logic component is ensured, preparing the component is the main task in the application engineering process. We would see if all components can be automatically generated through the current development.

The generators we need are

- inspection logic component generator which reads a state transition specification and creates an inspection logic component, and
- test case generator which produces test cases from the state transition specification.

We would define specifications to the generators in this development cycle for JCI.

6 Conclusion

We proposed the aspect-oriented software architecture for a code inspection tool. We have derived the following five crosscutting concerns other than object-oriented which is the primary concern:

- internal data model,
- language processing,
- data traverse,
- inspection logic, and
- decoupling the traverse and the internal data model access from the inspection logic.

We implemented the software architecture with design patterns. Composite, Interpreter, Visitor, Mediator, Command and State patterns are borrowed. The state transition model for the specification to the inspection logic is introduced. We demonstrated its usefulness and practicality of the approach with several examples. In discussion, we validated the usefulness of our aspect-oriented architecture, and the practicality of the specification method, Practice of PLSE with the software architecture is also discussed and we concluded that we have already prepared most of things we need for the PLSE practice.

Possible future research topics include design and implementation of:

- the inspection logic component generator, and
- test case generator.

We would define the specification to the generators in this cycle of JCI development.

Acknowledgement

This research has partially been supported by JSPS KAKENHI (Grant-in-Aid for Scientific Research (C) 22500037 and 24500049), by a grant from Canaly Research, Ltd. since 2007, and by Nanzan University Pache Research Subsidy

I-A-2 for the 2012 academic year. We gratefully acknowledge that ATM Japan, Ltd. and Canaly Research, Ltd. had put much effort to JCI's co-development in Project OCEAN (On the Job Learning Centered Education for Advanced engi-Neers) mainly by Nagoya University and Nanzan University under MEXT Leading IT Professionals Fostering Program since 2007 till 2009. We also appreciate co-operation with Canaly Research, Ltd. for the maintenance of JCI.

References

[1] Altho. JLint. [Online]. Available: http://artho.com/jlint/

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison-Wesley, 1994.

[3] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A System and Language for Building System-Specific, Static Analyses," in *Proc. ACM PLDI2002*. ACM Press, 2002, pp. 69–82.

[4] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," *ACM SIGPLAN Notices*, vol. 39, pp. 92–106, Dec. 2004.

[5] ISO/IEC, *ISO IEC 2511: Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models*, ISO.

[6] L. M. Northrop, "SEI's Software Product Line Tenets," *IEEE Software*, vol. 19, pp. 32–40, Jul. 2002.

[7] T. Osuka, T. Kobayashi, J. Mase, N. Atsumi, S. Yamamoto, N. Suzumura, and K. Agusa, "CX-Checker: A Customizable Coding Checker for C (in Japanese)," in *Proc. SES2009*. Tokyo, Japan: Kindaikagakusha, Aug. 2009, pp. 119–126.

[8] N. Rutar, C. B. Almazan, and J. S. Foster, "A Comparison of Bug Finding Tools for Java," in *Proc. ISSRE2004*. IEEE Computer Society, 2004, pp. 245–256.

[9] A. Sawada, M. Noro, Y. Hachisu, H. M. Chang, A. Yoshida, D. Osa, and A. Urano, "Design and Evolution of the Sftware Architecture for Source

Code Inspection Tools (in Japanese)," *Computer Software*, vol. 28, pp. 241–261, Nov. 2011.

[10] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[11] Sourceforge. (2011) CheckStyle. [Online]. Available: http://checkstyle.sourceforge.net/

[12] ——. (2012) PMD/Java. [Online]. Available: http://pmd.sourceforge.net/