On Aspect-Oriented Software Architecture: it Implies a Process as well as a Product

Masami Noro

Dept. Info. & Telecomm. Eng.
Nanzan University
27 Seirei-Cho, Seto
Aichi 489-0863, JAPAN
yoshie@nanzan-u.ac.jp

Atsushi Kumazaki

Dept. Info. Sys. & Quant. Sci.
Nanzan University
18 Yamazato-Cho, Showa-ku, Nagaya
Aichi 466-8673, JAPAN
d00bb002@iq.nanzan-u.ac.jp

Abstract

We assume that software architecture is a set of aspects which represent concerns multiple-dimensionally separated. Moreover, software architecture is assumed to be not just a product model but it also implies a software process for its implementation. We view the architecture as a set of aspects connected via join points. In this sense, the aspect is the composite component and the join point is the connector. The aspect, in turn, consists of a set of components from different abstraction level of the development stage. A connector implies an order of the development of connected components. Based on this assumption, the architecture (connectors) is to define a partial-order process for the development. This paper describes the idea which tells how a software architecture defines software process in the context of aspect-orientation. We demonstrated how it works with several examples.

1 Introduction

Software architecture has been a promising approach to software engineering problems[5, 14]. In these days, Post-Object Programming technologies[3], aspect-oriented programming in particular, is recognized as another promising approach to the problems[2]. Through the observation of several software development projects[10, 13], we realized that to apply an aspect-oriented concept to software architecture is not just yet another but one of "the" promising approaches to the problems.

We have demonstrated with several projects that a software architecture is not just a product model but implies a software process[9, 10]. In the projects, we found that a software architecture consists of a set of aspects which represent concerns multiple-dimensionally separated. In an aspect-oriented sense, a component of a software architecture is an aspect and a connector is a set of join points (pointcut). As a result, an aspect-oriented software architecture is not just a product model but implies a concurrent software process.

There are several related research projects on aspect-oriented software architecture[12] and aspect-oriented application frameworks[1, 11]. Navasa and Nakajima put much emphasis on construction process of architecture or framework. Constantinides gives an idea how to implement an application framework with a design pattern (Factory Method)[1]. These research projects are fruitful but they just talk about product view of software development. We also introduces architecture style employing aspect-oriented concept. It is not a yet another research on aspect-oriented software architecture because we also introduces process view of software development based on our architecture.

This paper is intended to describe what our definition of aspect-oriented software architecture is, to show how it implies a concurrent software process, and to discuss advantages of an aspect-oriented software architecture. In summary, an aspect-oriented software architecture implies a set of concurrent software processes which are for the development of each aspect. The processes are connected by the activity to implement a set of join points which associates the aspects. A case study to define an aspect-oriented software architecture for TCP/IP applications is given to validate our idea.

2 What is Software Architecture ?

From the experience obtained when we defined domain specific software architectures[9, 10, 13], we realized that a software architecture is not just a product model but also a container including a process planning scenario. The insight we gained meets the definition by the SEI discussion group in '94 of a software architecture that "the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time." This results in the paradox that a product defines a process, and we conclude from this that:

- shapes of components vary vertically across development stages,
- interrelationships (connectors) bridge between one stage and another, and
- they, in combination, implicitly show the order of component construction.

In Fig.1, we abstract a software architecture which defines a process. We assume object-oriented software development here. As shown in the figure, the software architecture includes components from the software model, the implementation stage, and the design result. The software architecture shown in Fig.1 indicates the process for developing the software abstracted in the architecture. We can easily see where design and implementation are needed component by component. The general scenario of the development can be stated as follows.

- Extract implementation-level component and plan to select the appropriate class or component library. Or, implement another subclass in a hierarchy.
- Implement classes and their hierarchy for the design-level component.
- Carry out the design and implementation for a component from the software model.

These three activities can be performed concurrently. Termination of all these activities triggers the coordination of the component. Then, we will proceed to testing and operation.

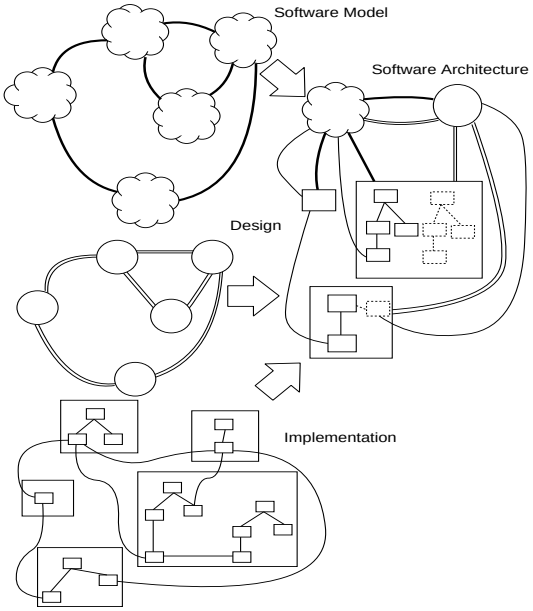


Figure 1: Software Architecture

2.1 Aspect and Architecture

In our definition of an aspect-oriented software architecture, a software architecture is composed of a set of aspects. In this sense, an aspect is a composite component and a join point is a connector. Here, "composite" means that an aspect is a set of components. An abstraction level of a component may differ from that of another component. As described above, there are several types (actually, a type of a connector is defined by abstraction levels of components the connector links) of connectors which define the development process. For the sake of simplicity, we do not assume aspects are nested. Fig.2 cartoons an example of our aspect-oriented software architecture.

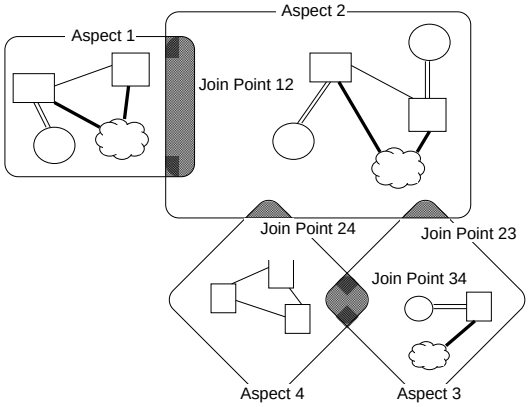


Figure 2: Aspect-Oriented Software Architecture

Once determined inter-aspect protocol in a pointcut, crosscut aspects can be developed in concurrent. That is, to define an inter-aspect protocol in join point12 of Fig.2 must precede the development

of Aspect1 and Aspect2. Two aspects can be concurrently implemented after the definition. The development process of each aspect, in turn, can be defined from hetero-links (connectors in different types) among components.

3 How Architecture Implies Process

As introduced in the previous section. hetero-links imply an "inner" software process for the development of an aspect. From aspect-oriented view, a join point is a connector and the relationship among aspects defines an "outer" software process. That is, the outer software process describes a partial order of aspect development, and another partial order of component development is given in the inner software process.

3.1 Primitive Rules to Decide Process

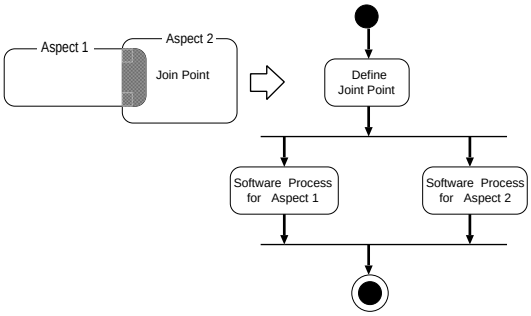


Figure 3: A Primitive Process-Derivation Rule for Aspects

For Aspects

If two aspects crosscut each other, inter-aspect protocol on the join point which associates the aspects must be defined first. Then, each aspect can be developed simultaneously. Fig.3 shows a primitive process-derivation rule for aspects. We borrow the activity diagram in RUP to describe a process.

For Components

For software process definition of components, an abstraction level is a key to derive the process. Here we assume that there are tree levels of abstraction: model, design, and implementation. A model-level component (a component from an abstraction level of

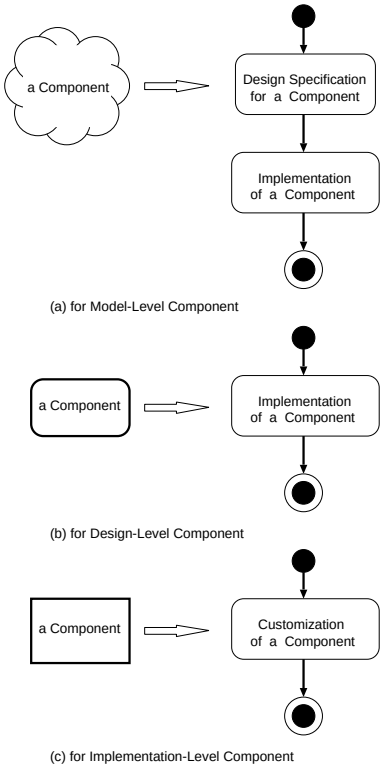


Figure 4: Primitive Process-Derivation Rule for Components

model) has neither its design specification nor its implementation. A design specification is given but its implementation is not done yet in a design-level component. Customization is needed, in general, for an implementation-level component.

To implement a model-level component, its design specification is needed first and then it is implemented. A design-level component is implemented on its design specification. An implementation-level component is customized if needed. These primitive process-derivation rules are shown in Fig.4.

3.2 Rules for Organizing Processes

There are nine ($3 \times 2 + 3$) use-relations (send-message-relations) among components in three levels of abstraction. Three relations are ones for the same level of abstraction (e.g. a model-level component uses another model-level component.) Other six are for components from different level of abstraction (e.g. a model-level component uses a design-level component and vice versa.) If a model-level component

(MC1) uses another model-level component (MC2), both MC1 and MC2 are required to give design specifications and their implementations. Since MC1 uses MC2, MC1's specification should be given first, and then MC1 and MC2 can be implemented. For MC2, it can be implemented after its design specification is given and MC1's specification is understood. Fig.5 shows the software process obtained from a relation in which a model-level component uses another model-level component. Software processes for other cases are also given in an Appendix. DC stands for design-level component and IC is for implementation-level component in the Appendix.

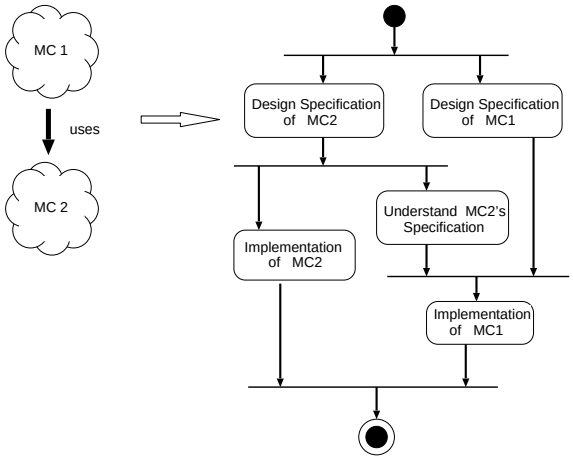


Figure 5: Software Process Organization Rule

In an aspect-oriented software architecture, software processes for the development of aspects are combined into a software process for the architecture. Definitions for inter-aspect protocols (join points) precede the processes for the aspects. If a software architecture consists of three aspects crosscutting one another and each aspect is composed of components as in Fig.6 (a), the software process will be one shown in (b).

4 Aspect-Oriented Software Architecture for TCP/IP Application

To demonstrate that our idea of aspect-oriented software architecture is reasonable, we show an example of TCP/IP application for which we have constructed the software architecture[10]. In our previous architecture, an object-oriented concurrent model was adopted for TCP/IP applications. We modeled a TCP/IP application as a set of concurrent objects on isolated machines. The objects was conceived as state transition machines. The resultant architecture was domain-specific layered model.

Through the construction of a software architecture for TCP/IP applications, we realized that several concerns crosscut more than one layer. They are security, input handling, and so on. In the following sections, we treat those concerns as well as component of dominant decomposition as aspects in thr aspect-oriented software architecture for TCP/IP applications.

4.1 Three Views of Software Architecture

To represent a software process in a software architecture, our software architecture includes the following three views: abstract view, concrete view and process view. It is an improved and revised version of Kruchten's definition[8]. The reason why we did not use Kruchten's definition of an architecture is that his definition assumes the same level of abstraction for the components of the architecture and a single type of connection among the components.

In our model, the abstract view corresponds to component-connector model of the software architecture. As mentioned earlier, a component comes from one of any development stages, while a connector vertically and/or horizontally ties components together.

The concrete view identifies abstraction level of a component. It shows component's abstraction level by presenting what kind of technique can be applied to design and/or implement a component. As a result, connectors would be hetero-links which are keys for defining a software process.

The software process to implement the architecture being defined is derived from concrete view. Process view holds the process that shows the partial order of component development.

We also uses these three views for defining an aspect-oriented software architecture.

4.2 Aspects in TCP/IP Application

Here, we describe just the aspect-oriented software architecture for a client of a TCP/IP application to save the space without loss of generality.

The following concerns are recognized through observing the implementation of the layered architecture. Since a TCP/IP client is generally an interactive software, to use the MVC architecture[6] is natural idea to design and implement. In the MVC architecture, Model, View, and Controller are major components and they are components realizing separated

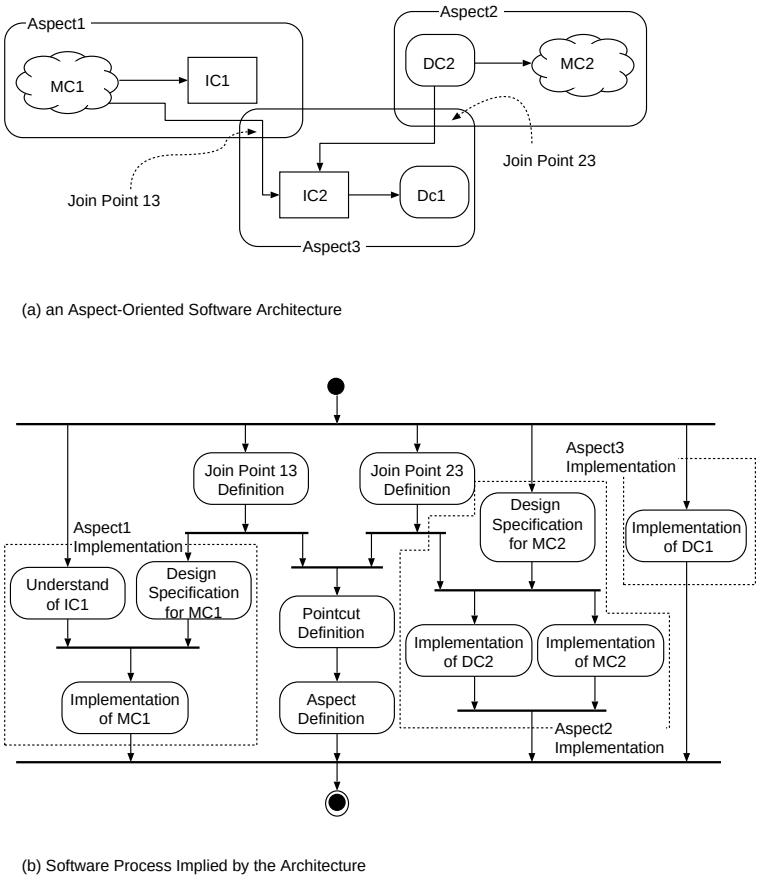


Figure 6: Aspect-Oriented Software Architecture and its Development Process

concerns. Controller and View are also concerns in our aspect-oriented architecture. Model is further decomposed into multiple concerns.

- Controller: This concern is about MVC's Controller.
- View: This is MVC's View.
- Input handling:

  A TCP/IP client has two ports to accept data input, a standard input and a communication socket. This concern is about how to handle those two ports: doing selective wait, implementing multiple threads for the ports, or so forth.

- State transition machine:

  Since we assume that a TCP/IP application is a set of concurrent objects which realize state transition machines, how to describe state transition machine is a major concern in its development.

- An action corresponding to an event:

  How actions are realized is one of concerns whether or not it is separated.

- Remote object access: A method for realizing remote object access is a concern.
- Data communication:

  This concerns about what kind of data communication protocol to be implemented. It should be compatible to existing TCP/IP applications, or it is not.

- Error processing:

  This concern is on how errors such as timed-out during data communication are implemented.

- Efficiency:

The efficiency concern is on a non-functional requirement. This is about how data communication is implemented in an efficient way at running time.

- Security:

  It is another concern on another non-functional requirement. How to implement a secure TCP/IP application is this concern.

Fig.7 shows these concerns, aspects, and their relationships in a TCP/IP application. As in the figure, an aspect is a container to implement several concerns (a single aspect may be used to realize a single concern.) Concerns are collected into a single aspect if concerns do not crosscut one another. Concerns in an aspect are related one another. To relate means that one of components which realize a concern uses (sends a message) to one of other components implementing another concern, or vice versa. An arrow in the figure means that the component(s) implementing a concern which is a source of an arrow uses one of the components which realize a concern at the destination of the arrow. If multiple arrows converge on a concern, the concern is crosscut.

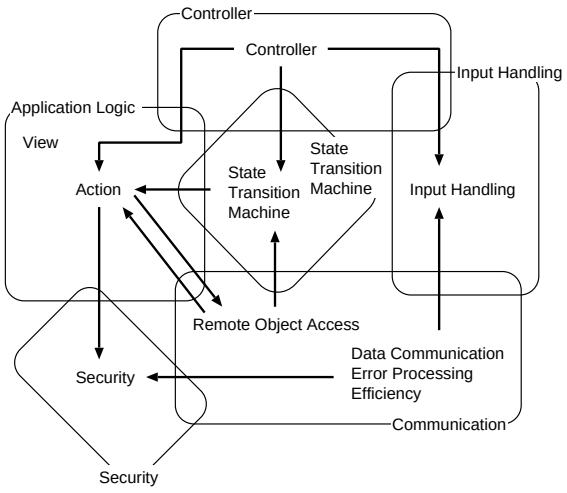


Figure 7: Concerns, Aspects, and their Relationships in a TCP/IP Application

4.3 Abstract View of TCP/IP Application Architecture

Our aspect-oriented software architecture for a TCP/IP application is aspect-oriented version of the architecture we previously constructed[10]. We borrow portions of the previous architecture to draw the aspect-oriented architecture.

Fig.8 outlines a part of our aspect-oriented architecture for a client. `StdIn` in the figure implements a class for standard input . Class `Com` encapsulates a user input and is passed to `ComResTable`. `ComCreator` makes an instance of `Com` and passes it along to `ComResTable`. `ComResTable` has to have a possible set of `Com` and `Action`. In fact, the table includes the algorithms for creating an `Action` object rather than simply action itself. We designed it this way because the algorithm for the creation procedure is application dependent. The strategy pattern is for storing an algorithm in an object. We use the pattern to implement `Strategy` which makes an instance of `Action` from a `Com` object. `STM` stands for a state transition machine and holds multiple `State`s. `InputHandler` implements selective wait and character read.

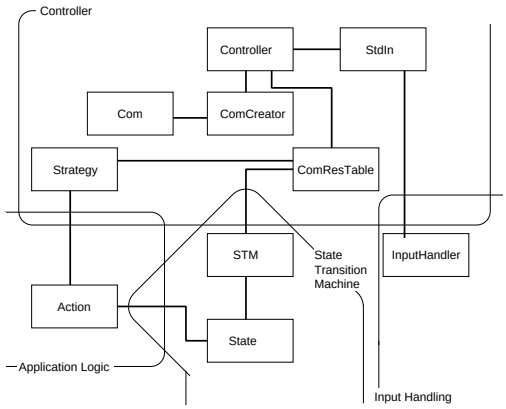


Figure 8: Part of Abstract View of TCP/IP Client Architecture

4.4 Concrete View of TCP/IP Application Architecture

The concrete view defines the implementation techniques that must be employed for the development of software based on the architecture. That is, the view indicates what kind of techniques must be used for

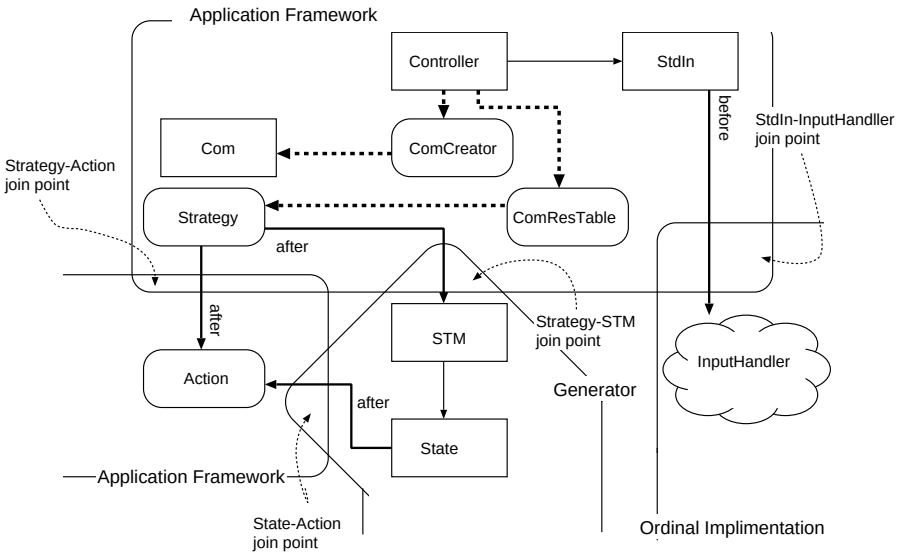


Figure 9: Part of Concrete View of TCP/IP Client Architecture

the implementation of the components in the abstract view.

The controller aspect and application logic aspect include application dependent objects. Since they are heavily application-dependent, we apply the technique of application framework for the implementation of these two aspects. Since a hotspot of a framework has design specification but its implementation, a design-level component corresponds to the hotspot.

State transition machine can be generated from its specification. We could find out such generators here and there. Since the specification given to the generator could be graphical and understandable, they prefers the generator to other development techniques such as libraries, application framework, *etc.* Generated components are implementation-level component since they do not need further code writing. We assume that generated codes are written in the state pattern[4].

Input handling aspect is platform-dependent. We are forced to design and implement it from scratch.

Fig.9 sketches a part of TCP/IP application architecture's concrete view. It shows abstraction levels to the components and arrows which represent development process.

Assume that class `Strategy` has a method `createAction` which is for making a new `Action` instance, and class `State` has a method `trans` for state transition, and `Action` has a method `doIt` to invoke it.

The aspect `ApplicationLogic` may be described as the following if we borrow the notation of AspectJ[7].

```
aspect ApplicationLogic {
  pointcut create(Strategy s) :
    instanceof(s) &&
        receptions(Action createAction(s));

  after(Strategy s) returning(Action a) :
      create(s) {
    a = new Action(s);
  }

  pointcut fire(State s, Action a) :
    instanceof(s) && receptions(void trans(a));

  after(State s, Action a) : fire(s, a) {
    a.doIt();
  }
}
```

4.5 Process View of TCP/IP Application Architecture

Fig.10 shows a aprt of development process which can be drawn from the concrete view in RUP.

4.6 Implementation

There are several ways to implement an aspect-oriented software architecture. One of the most natu-

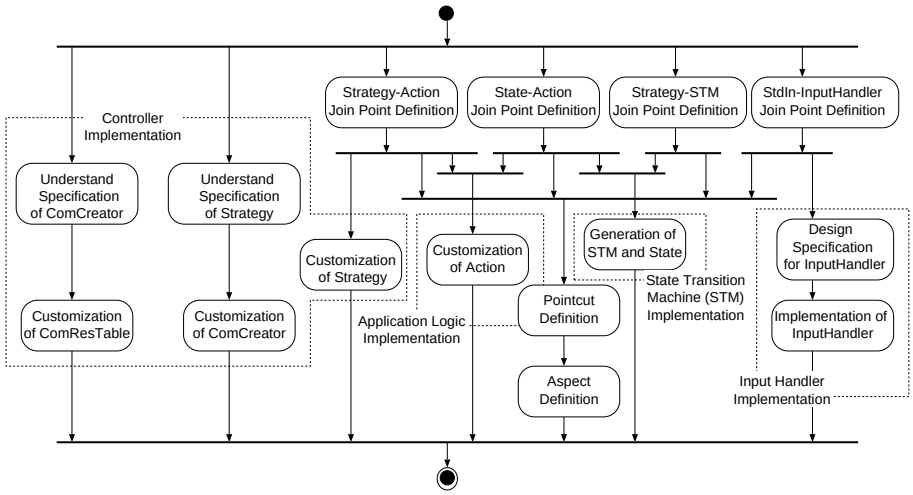


Figure 10: Part of Process View of TCP/IP Client Architecture

ral ways is to implement in aspect-oriented programming languages such as AspectJ, AspectC++[15] and so on. However such implementation would be language dependent.

Instead, we have implemented our aspect-oriented TCP/IP application architecture with Factory Method as in [1]. This way of implementation does not lose language independence and as a result it preserves practicality with state-of-the-art (aspect-oriented) technology.

5 Discussion

The advantage of aspect-oriented programming is that we can separately implement crosscutting concerns. The implementation would be flexible enough to do away with future maintenance. The advantage is, however, in the programming language level. Software architecture, on the other hand, contributes to get shortcut to the development. We could skip requirements definition and design of a system. There is, however, a problem that we cannot always find out an orderly-fashioned software architecture in all application domains. Non-functional requirements almost always lead the architecture to ill-structured one. To apply the aspect-oriented concept to earlier stages enables us to encounter an aspect-oriented software architecture. With the aspect-oriented concept, we cloud find out a sound architecture for the domain where we could not. The case study in the previous section tells us the limitation of layered architecture and the powerfulness of aspect-oriented software architecture.

In addition to the advantages obtained by the marriage of aspect-oriented concept and software architecture, we could get an advantage on software process. We have proposed that a software architecture is not just a product model but implies its software process[10]. To assume a software architecture as a set of components having different abstraction levels and different types of connectors yields a partially-ordered process for software development. A concurrent software process whose subprocesses are also concurrent could be drawn with an aspect-oriented software architecture as described in the previous section.

6 Conclusion

We introduced our idea on aspect-oriented software architecture. We have shown that the aspect-oriented software architecture gave us a well-formed software architecture of what we could not get without the aspect-oriented concept.

A software architecture implies a software process if the architecture is defined as a set of components connected by different type of connectors in different level of abstraction. A concurrent software process can be implied by an aspect-oriented software architecture which is, in our definition, a set of aspects representing concerns multiple-dimensionally separated.

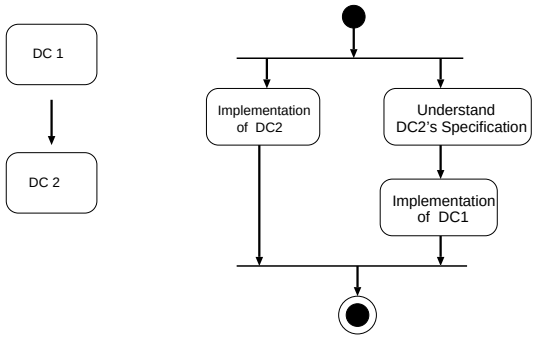
Future research topics on the aspect-oriented software architecture include the following.

– To apply our idea of aspect-oriented software architecture to other domains.

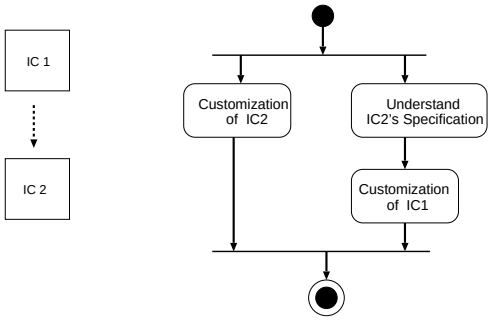
  We are now constructing another domain-specific aspect-oriented software architecture for a Web-based information system. The architecture would sort unordered Web-related software development techniques and would give us concurrent software process for the WIS development.

– To design and implement an aspect-oriented architecture centric software development environment.

  A software process is a key to the integration of software tools in a development environment. An aspect-oriented software process, in our definitions, implies a concurrent software process. Those mean that we could integrate software tools on the aspect-oriented software architecture. We draw a big picture in which multiple development personnel check in and out software repository reflecting the architecture. The process scenario is embedded in the mechanisms on checking in and out.



(b) A Design-Level Component Uses another Design-Level Component



(c) An Implementation-Level Component Uses another Implementation-Level Component

Appendix : Software Process Organization Rules

Acknowledgements

Here we thank our colleague, Takeshi Tomonaga, who helped us to format diagrams in the paper.

References

[1] C. Constantinides, A. Bader, T. Elrad and M. Fayad "Designing an Aspect-Oriented Framework in an Object-Oriented Environment," *Computing Surveys 32, 41*, 2000.

[2] T. Elrad, R. E. Filman, A. Bader, Eds., *Special Issue on Aspect Oriented Programming*, CACM, Vol. 44, 2001.

[3] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison Wesley, Boston, 2000.

[4] E. Gamma, *et. al*, *Design Patterns*, Addison Wesley, 1995.

[5] D. Garlan and D. E. Perry Eds., *Special Issues on Software Architecture*, IEEE Trans. Soft. Eng., Vol. 21, No. 4, 1995.

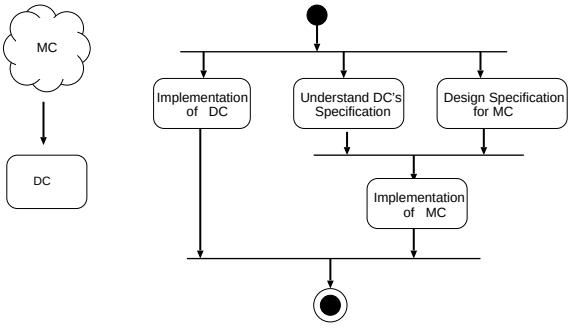
[6] G. E. Krasner and S. T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *J. of Object-Oriented Program*, Vol. 1, No. 3, pp.22-49, Aug./Sep. 1988.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold,"An Overview of AspectJ", in *Proceedings of the European Conference on Object-Oriented Programming(ECOOP)*, Springer-Verlag. Hungary, 2001.

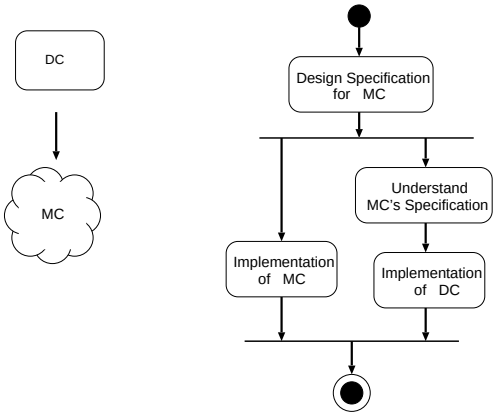
[8] P. B. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, Vol. 12, No. 6, pp.42-50, Nov. 1995.

[9] A. Kumazaki, M. Noro, H. Chang and Y. Hachisu, "A Software Architecture for Web-based Information Systems," (in Japanese) to appear in *Proc. of IPSJ Object-Oriented Symposium 2002*.

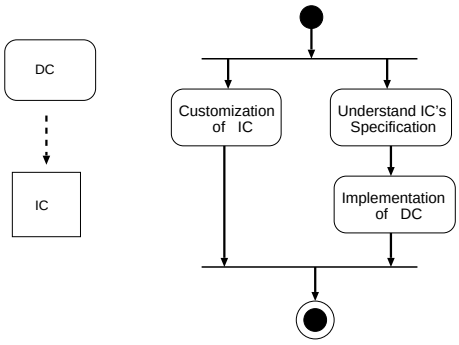
[10] A. Kumazaki, M. Noro, H. Chang and Y. Hachisu, "An application Framework for TCP/IP Applications," to appear in *Proc. of COMPSAC2002*.



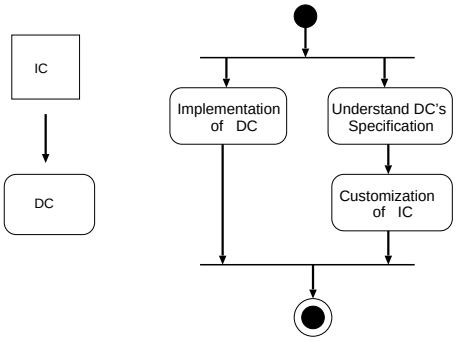
(d) A Model-Level Component Uses a Design-Level Component



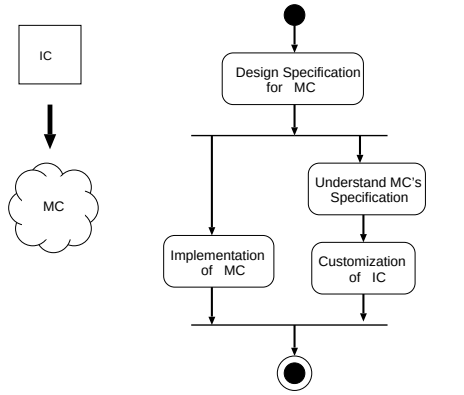
(e) A Design-Level Component Uses a Model-Level Component



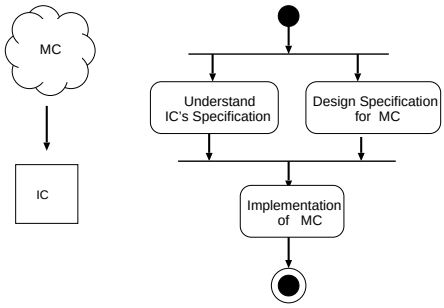
(f) A Design-Level Component Uses an Implementation-Level Component



(g) An Implementation-Level Component Uses a Design-Level Component



(h) An Implementation-Level Component Uses a Model-Level Component



(i) A Model-Level Component Uses an Implementation-Level Component

[11] "Separation of Concerns in Early Stage of Framework Development," Workshop on Multidimentional Separation fo Concerns, *OOPSLA2001*.

[12] A. Navasa, *et al.*, "Aspect Oriented Software Architecture: a Structural Perspective," http://wwwhome.cs.utwente.nl/~bedir/Research.htm.

[13] M. Noro, "A Software Architecture for Vending Machine Control," *Proceedings of ISFST '97*, Nov. 1997.

[14] M. Shaw, and D. Garlan, *Software Architecture, Perspective on an Emerging Discipline*, Prentice Hall, 1996.

[15] O. Spinczyk, *et al.*, "AspectC++: an Aspect-Oriented Extension to C++," *Proc. TOOLS Pacific 2002*.