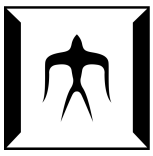

A Dissertation Submitted to Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of Doctor of Science in Mathematical and Computing Sciences

---

An Aspect-Oriented Programming Language for Agile Software Development

Muga Nishizawa

Dissertation Chair:
Shigeru Chiba

March 2008,

Abstract

This thesis proposes two new language constructs for aspect-oriented programming (AOP) and a new AOP language that provides those language constructs. AOP is useful for agile software development. However, existing AOP languages have not been used on agile software development yet. This is because existing AOP languages are not good enough for agile software development. This thesis first analizes the features of agile software development and shows that AOP is useful for agile software development with two example scenarios. Then it explains that AspectJ is not good enough for concurrent development by several software engineers and for developing distributed software in the agile methods. AspectJ is one of existing general-purpose AOP languages for Java.

To solve the problems of existing AOP languages such as AspectJ, this thesis presents our AOP language named GluonJ. GluonJ is based on AspectJ. It provides mechanisms for pointcut-advice and inter-type declarations. Moreover GluonJ provides new language constructs for AOP named dynamic refinement and remote pointcut. Dynamic refinement allows software engineers to change the behavior of an existing class according to dynamic contexts. By using dynamic refinement, a software engineer can implement his assigned feature as an aspect without consideration of other features that other software engineers are developing at same time. Then he easily merges his implemented feature (his aspect) into an existing program. Also remote pointcut allows modularizing crosscutting concerns in distributed software as simple aspects. Although AspectJ can separate distributed crosscutting concerns from the rest of the program as aspects, the aspects consist of several complex sub-modules. By using remote pointcut, a software engineer can write a

feature that is newly appended to distributed software as a simple aspect. This aspect is a single component at the code level and does not include explicit network processing such as Java RMI.

Acknowledgments

This work would not have been possible without the guidance and friendship of my superviser Shigeru Chiba. He supported me seven years from a bachelor student to a PhD student in Tokyo Institute of Technology. He gave me the freedom and the means to mature my ideas while always pointing me in the right direction. I express my deepest gratitude to him.

I deeply thank the current members, alumni, and alumnae of CSG (Chiba Shigeru Group). In particular, from Kenichi Kourai, Michiaki Tatsubori, Daisuke Yokota, Yoshiki Sato, Romain Lenglet and Yoshisato Yanagisawa, I got precious comments that greatly improved the quality of my thesis. Also I greatly thank my thesis committee, Masataka Sassa, Satoshi Matsuoka, Ken Wakita, Osamu Watanabe, and Etsuya Shibayama. They gave me continuous feedback during the writing of this thesis.

Finally, I want to thank my mother, father, and sister for support and care during the years of my study.

I am grateful for financially support received from Japan Science and Technology Agency and from The Information Technology Promotion Agency, Japan.

Muga Nishizawa
March 2008

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently, the methods for agile software development have been getting widely accepted in software industry. The agile methods are ones categorized into the methodology called iterative software development. However, unlike existing methods such as waterfall and iterative models, they have high flexibility against requirement change. The requirements by customers are often ambiguous and are changed during the software development. In the agile methods, software engineers divide the total period of the software project into a large number of short iterations (short periods). They incrementaly develop the software the five phases of requirements analysis, design, implementation, testing, and documentation by iterating. At the begining of each iteration, they analize the requirements from their customers and then improve the software product obtained after the previous iterations to satisfy those requirements. Several different methods for agile software development have been proposed and presented before.

1.1 Motivation

Existing methods for agile software development allow software engineers (or project teams) to develop software products more rapidly. On the

other hand, existing programming languages are not enough for agile software development.

1.1.1 Features of Agile Software Development

We analized the features of existing agile methods before developing an ideal language suitable for agile software development. The following is a list of the results of our analysis.

1. Analizing requirements in each iteration
2. Removing features from an existing program
3. Refactoring of an existing program
4. Concurrent development by several engineers

First, requirements analysis in each iteration is the most significant feature of agile software development. Software developers show their customers a working software product at the beginning of each iteration. By checking the working program, the customers can easily notice requirements that they want. The customers can give their developers clear requirements. This requirements analysis is iterated many times in the total period of software development. Unlike existing methods of waterfall model, software engineers would thus be able to develop a software product that their customers want. As a result, agile software development can help increase the flexibility and productivity of a software program.

Second, software developers ask for feedback (requirements) of a software product from their customers many times on agile software development. The requirements of customers often include the deletion of an existing feature that had been developed by the previous iterations. Software engineers must quickly respond to such a requirement and then delete the features.

Third, software developers often refactor a software product developed by the previous iterations before they design and implement new features that are required by the current iteration. In an agile method, software developers append new features to the program of a software product and its specification step by step in each iteration. The program that has been developed in this way often makes it difficult for software

developers to add new functions. Thus, since software developers increase the modularity of its software product, they sometimes refactor the software product. Refactoring can help software developers easily append new function to their software product.

Forth, rapid development of a required software product is the aim of any methods of software development process, not just one of the agile methods. To develop a software product rapidly, more software engineers cooperate with each other, divide the tasks and burdens appropriately, and then develop it. In the case of agile software development, more software engineers divide newly appended functions appropriately.

1.1.2 Features of an Ideal Programming Language

By using the above results of our analysis, we consider an ideal programming language for agile software development. The ideal language should have four features as follow.

(A) Separation of features

(B) Without editing an existing program

(C) Concurrent development

(D) A statically typed language

First, an ideal language for agile software development should satisfy separation of features. As mentioned above, requirements of customers are based on features (functions) step by step in each iteration on agile software development. For example, the customers require their software engineers to add a new feature to an existing program, remove a feature from the existing program, and improve an existing feature in the program. Thus software engineers should focus on a appended feature only and be able to develop the implementation of it. Unfortunately, most of existing programming languages are not enough for separation of features. The implementation of a newly appended feature in an existing language often cut across several parts of the existing program. Thus It is difficult to focus on the feature only.

Second, an ideal language should allow software engineers to improve an existing program without editing the program. Customers often require the deletion of features that had been implemented by the previous iterations from a software product. Software engineers must quickly

add/remove the specified functions to/from the program according to requirements of their customers. However, most of existing languages do not enable improving an existing program without editing the program.

Third, an ideal language should enable several software engineers to easily develop several new features in parallel at same iteration. To develop a software product more rapidly, several developers divide their tasks appropriately. Since requirements of customers are based on features on agile software development, an ideal language should allow several software engineers to implement different features at same time. The ideal language should easily enable to marge the developed functions into the existing program.

Fourth, an ideal language should be a statically typed language. On concurrent development by several software developers, the types of modules in a software product should not be changed. It is because those types can be used by several software developers as the specifications of the modules. The implementation of a newly appended function often cut across several modules in an existing program. If the types (specifications) of the modules is not changed, a developer who implements the feature can safely use the modules. On the other hand, dynamic languages enable changing the type of a module at anytime. To develop a function, a software engineer of its function must carefully use several modules that have already been developed since the type of module may be changed at runtime. If he does not notice that the type is dynamically changed, the instance of the type would cause some errors in the function that he implements. A statically typed language also allows a software engineer to enjoy several support provided by IDEs (Integration Development Environments).

1.2 Our New AOP Language

In this thesis, we propose our new AOP language named *GluonJ*. It is more suitable than existing programming languages for agile software development. It is based on AspectJ, which is one of existing general-purpose AOP languages. GluonJ provides the mechanisms for pointcut-advice and inter-type declarations, which is called refinement here. Altough AspectJ satisfies the above features of an ideal language for agile software development to a certain degree, it is not enough for the features (A) and (C). To resolve the problem of AspectJ, we thus propose

two new language constructs for AOP: *dynamic refinement* and *remote pointcut*. We have appended these language constructs to GluonJ.

The reason why GluonJ is based on AspectJ is because AspectJ is the most suitable language for agile software development in existing programming languages. AspectJ satisfies the four features of an ideal language for agile software development. It allows software engineers to separate newly appended features to the existing program as aspects to a certain degree. The aspects can easily be added/deleted to/from the program. By using cflow that AspectJ provides, a software engineer can develop his part independently of other members' parts. Moreover AspectJ is an extension to Java, which is a statically typed language. Thus we extended AspectJ for agile software development and developed GluonJ.

GluonJ is an extended Java language. One of our contributions is the pragmatic design of GluonJ's language construct for AOP. Our extension to Java is small. GluonJ uses annotations and thereby does not extend the lexical syntax of Java. It exploits the type system of Java as much as possible. Thanks to these, a GluonJ program can be developed on a normal Java IDE (Integrated Development Environment). Particularly, software engineers can enjoy the coding supports by the normal IDE even for GluonJ programming. A GluonJ program is compiled by a normal Java compiler. Only a special runtime system is needed to run a GluonJ program. We introduced these features for industrial acceptability, where software engineers tend to stay with existing tools.

1.2.1 Dynamic Refinement

We propose dynamic refinement, which allows software engineers to dynamically refine the definition of an existing class. By using dynamic refinement in GluonJ, software engineers can redefine existing methods and append new methods, fields, and interfaces to an existing class according to dynamic contexts. Since these changes are described in a separate component (or module), this language mechanism is useful for separation of concerns.

GluonJ that provides dynamic refinement satisfies the feature (C) of an ideal language for agile software development in the previous section. When several software engineers append different new features to an existing program in parallel, dynamic refinement allows a software engineer to separate his features from the rest of the current program. Moreover

he can write his implementation without consideration of other features that other engineers are developing.

The concept of dynamic refinement is similar to the mechanism of dynamic scoping. Generally speaking, a language that has dynamic scoping is not easy-to-use. Such a language decreases the readability and understandability of its program. However, when software engineers improve a software product on agile software development in parallel, dynamic scoping is often effective. Thus GluonJ exploits dynamic scoping.

Although GluonJ allows software engineers to dynamically appending a method to an existing class, a GluonJ program never throws a NoSuchMethodException if it is successfully compiled and loaded. A naive implementation of dynamic refinement would wrongly allow a client to call an unavailable method, which will be appended later by refinement but not now. To avoid such a wrong call, which will throw a runtime exception, GluonJ requires programmers to follow some programming conventions. A GluonJ program satisfying these conventions never calls an unavailable method. If a program does not satisfy the conventions, it is statically detected before the program starts running. To do this, GluonJ exploits the type system of Java and the class loader of GluonJ.

GluonJ allows software engineers to dynamically refine an existing class during runtime to a certain degree. It allows applying and removing refinement to/from a class on demand. Naively designed dynamic refinement may allow a call to a method that has not been appended yet or that has been already removed and then it may cause a runtime type error. However, a GluonJ program never causes such a runtime type error as a NoSuchMethodException although it may fail an explicit type cast and throw a ClassCastException. To guarantee this property, GluonJ does not use a custom type checker. It exploits the type checking by a normal Java compiler and the verification by the custom class loader of GluonJ.

1.2.2 Remote Pointcut

We propose remote pointcut, which is new AOP language constructs for distributed computing. A remote pointcut is a function for designating join points in the execution of a program running on a remote host. Although a pointcut in AspectJ identifies execution points on the local host, a remote pointcut can identify them on a remote host. In other words, when the thread of control reaches the join points identified by a

remote pointcut, the advice body associated with that remote pointcut is executed on a remote host different from the one where those join points occur. Remote pointcuts are analogous to remote method calls, which invoke the execution of a method body on a remote host.

GluonJ that provides remote pointcut satisfies the feature (A) of an ideal language for agile software development in the previous section. This language construct can simplify the code of a component implementing a feature in distributed software as an aspect. AspectJ is a useful programming language is a useful programming language for developing distributed software. However, even if a feature can be implemented as a single aspect at the code level, it might need to be deployed on different hosts and it would therefore consist of several sub-components or subprocesses running on each host. Remote pointcut enables implementing such a feature as simple, non-distirbuted component without concerns about network processing.

The aspect weaving in GluonJ is performed at load time on each participating host. The normal Java classes on each participating host must be loaded by the class loader provided by GluonJ. This class loader weaves aspects and classes on the fly. The compiled aspects are stored in the aspect server. The parts of the compiled code except for the advice bodies are automatically distributed by the aspect server to each host, so the latest aspects can be woven when the classes are loaded. The software engineers of GluonJ do not have to manually deploy the compiled aspects to every host.

1.3 The structure of this thesis

The structure of the rest of this thesis is as follows:

Chapter 2

Chapter 2 first explains agile software development as technological background of this thesis. It shows that software engineers can rapidly develop a software product by using existing methods of agile software development. Then we show that existing programming languages are not good enough for agile software development and present the four features that an ideal language suitable for agile software development should have.

Chapter 3

Chapter 3 proposes a new AOP language for agile software development, named GluonJ. Alough GluonJ is based on AspectJ and provides language constructs for pointcut-advice and inter-type declaration named refinement, it has two new language constructs for AOP. This chapter presents the motivation of these language constructs simply. Then we illustrate an overview of pointcut-advice and refinement in GluonJ in detal.

Chapter 4

Chapter 4 proposes dynamic refinement, which allows software developers to changing the behavior of an existing class according to dynamic contexts to a certain degree. It can dynamically redefine methods and append a new method to the class. Chapter 4 also presents new AOP language named Dynamic GluonJ for Java. It provides a language construct for dynamic refinement. Moreover, a Dynamic GluonJ program never throws a runtime exception reporting that an undefined method is called. Guaranteeing this fact is not straightforward because Dynamic GluonJ allows programmers to refine a class definition during runtime.

Chapter 5

Chapter 5 proposes remote pointcut, which is a language construct for distributed AOP. Remote pointcut allows modularizing a crosscutting concern in distributed software as a simple aspect. By combining an AOP based agile method and remote pointcut, engineers efficiently develop a distributed software product rapidly and easily. Moreover Chapter 5 presents new AOP language named Remote GluonJ for Java. It provides remote pointcut.

Chapter 6

Finally, this thesis is concluded in Chapter 6. This thesis presents contributions of it and future directions.

Chapter 2

Limitations of Existing Programming Languages on Agile Software Development

A wide variety of methods of software development has been proposed or presented before [66, 78, 34, 18, 10]. Most of such methods are methodologies for managing software development projects. Due to diverse factors, a software development project easily misses a development deadline and then runs over budget. However, project management allows reducing the delivery delay or over budget of its software project. It can improve the productivity and quality of a software product. Thus some of those methods have been getting widely recognized in software industry today.

In 1970 Royce proposed the initial concept of waterfall model [66, 67], which is well-known software development process[1]. Software engineers by a method with this model sequentially develop a software product through the phases of requirements analysis, design and implemention of a product, testing programs, and installation and maintenance like a waterfall.

[1] In that paper the term "waterfall" was not used.

According to Royce's original waterfall model, software engineers proceed from one phase to the next phase in a sequential order. For example, software engineers first start the phase of requirements analysis. When they fully finished analizing the requirements of their customers, they start the phase of design. When the design is fully complete, they start the phase of implementation of that software product.

Existing methods of waterfall model do not allow software engineers to return to previous phases that they have already finished. For example, after software engineers analize the requirements of their customers, they start the next phase according to the result of the analysis. When they design and implement a software product, they cannot return to the phase of requirements analysis and then change the result of the analysis. Thus they must completely finish each phase. Note that, the current version of waterfall model enables software engineers to return from one phase to only the previous phase.

However, existing methods of waterfall model do not often allow software engineers to develop a software product that their customers want. This reason is because the needs of their customers are ambiguous. The paper-based analysis of the requirements are not enough for designing and implementing a software product that the customers want. When the engineers installed the developed product on the environment of the customers, the customers first review the working software product and then notice their real requirements. Although the engineers must improve the software product according to newly appended needs of their customers, it is difficult to rapidly add newly appended functions to the program that was almostly developed. As a consequence, the delivery delay is caused.

2.1 Agile Software Development

To address the problem of the existing methods such as waterfall model, *agile software development* has been proposed [37, 10]. It is one of methodologies for rapidly developing a software product. It was proposed in 1990s as part of a reaction against heavyweight software development such as a method with waterfall model. Thus agile methods are called lightweight methods. In 2001 the organization named *Agile Alliance* [1] promoted agile software development. At same time, that members of the organization were published an *agile manifesto*, which

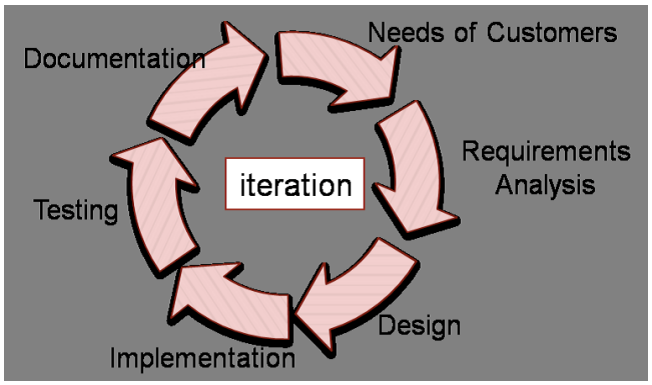

Figure 2.1. software development cycle by one iteration on agile software development

is widely regarded as the canonical definition of agile development, and accompanying agile principles [8]. The concept of agile manifesto is the following:

- *Individuals and interactions* over processes and tools.
- *Working software* over comprehesive documentation
- *Customer collaboration* over contract negotiation
- *Responding to change* over following a plan

Several agile methods respect software development with iterative model that was proposed in 1980s. In software development with an agile method, software engineers divide the total period of the development into a large number of short periods, named *iterations*. The iteration is from two or three weeks from two or three months. They develop a software product in an iteration. At the beginning of an iteration, they get the requirements of their customers. They analize the requirements. According to the result of the analysis, they design, implement, test a software product. At the beginning of the next iteration, they show their customers the developed program and receive the requirements of their customers again.

The significant feature of an agile method is that software engineers finish developing a software product that their customers want in each iteration. At the end of each iteration, they complete a working software

according to the result of requirements analysis. Unlike paper-based requirements analysis, it is easy for their customers to send the engineers the requirements, by showing the working software product. The requirements of their customers are more clear and concrete. Thus it is not difficult for the engineers to analize the customers' requirements.

Extreme Programming Extreme programming is a well-known method of agile software development [37]. In the late 1990s, it was created and proposed by Beck. Although extreme programming is one of agile methodologies, it is famous as a set of practices for agile software development such as test-first development, pair programming, and refuctoring. Most of these practices have been proposed before.

Extreme programming is effective when the requirements of customers are often ambiguous. One iteration in extreme programming is one (or two) week. For example, at the begin of one iteration, an engineer receives feedback from his customer and develops the working software according to the feedback in a week. At the begin of the next iteration, he gives his customer feedback again. And then according to the feedback, he improves the software that has been developed in a week.

Rational Unified Process (RUP) Rational unified process was created by the Rational Software Corporation in the 1980s and 1990s, a division of IBM since 2003 [10]. The rational unified process is a set of concrete methods of object-oriented software development. The teams (or developers) of software projects can select which methods they should use for the software development. Thus the rational unified process is called a framework for creating concrete processes.

The rational unified process is also one of methods of iterative software development process. It allows developers to improve that their software product in each iteration. Generally speaking, in the rational unified process, developers iterate requirements analysis of their software product from three to six times total a development cycle. The number of iterations is desided according to business case and risk list.

2.2 Features of an Ideal Language for Agile Software Development

Today software engineers has been developing products by the combination an existing programming language and agile software development. However, existing languages are not enough for software development by agile methods. To propose a new language suitable for agile software development, we surveyed the features of agile software development. And then according to the results of the survey, we considered the features of an ideal language for agile software development. The features are following.

(A) Separation of features

(B) Without editing an existing program

(C) Concurrent development

(D) A statically typed language

First, an ideal language suitable for agile software development should allow software engineers to write one newly appended function as one module. In an agile method of software development, the requirements of customers are often based on functions (features). For example, the customers require their developers to add a new function to the existing program, remove an existing function from the program, and improve the existing function.

Software engineers should consider a required function only and then be able to develop its implementation. If they implement a newly appended function in most of existing OOP languages, the implementation of the function often cut across several programs in an existing software product. To implement the function, they must edit the several programs. In this case, it is difficult for them to consider the function concern only. Rather, they need to consider not only the concern but also the concerns of the several edit programs.

To append a new function, software engineers with an agile method often refactor an existing program in advance. This reason is because it is not easy to append the implementation of the function to the existing program. Although refactoring of an existing program is necessary for improving the maintainability of the program, software engineers involve

an immense amount of time and effort to do it. If a language satisfied the feature (A) and enables separating a new function from the rest of the program as a module, they would not refactor the program. They can save the trouble of refactoring. Thus the total efficiency of the software development increase.

Second, an ideal language should enable implementing a newly appended function without editing an existing software product. At the beginning of an iteration, according to the requirements of customers, software engineers often delete an existing function from a program that has been developed by the previous iterations. Since a function written in an existing language cut across several programs in a software product, software developers must edit the several programs for deleting the function. If a language satisfies the feature (B), it is easy for software engineers to delete a function from the rest of the program.

Third, a language suitable for agile software development should enable software engineers to easily develop several new functions in parallel at same time. At the beginning of one iteration, customers often require software engineers to append several new functions to an existing program that has been developed by the previous iterations. According to such needs of customers, software engineers must rapidly be able to implement several functions and then merge those implemented functions into the existing program.

Concurrent development of a software product is one of significant issues in any methods of software development. To develop a software product more rapidly, software engineers divide the tasks and burdens appropriately. In a method of waterfall model, the result of requirements analysis is not changed during software developemnt. The specification of a software product is not changed during that period. Thus software engineers of waterfall model divide a software product into several components in the product consists and then develop different components each other. Since the specification of those components is not changed, a software engineer is not affected by the progress of other engineers.

Unlike waterfall model, the requirements of customers will be changed at every iteration under agile software development. Software engineers of an agile method analize the requirements according to the needs of customers. They cannot divide a software product into several components and develop different the components each other. It is because the specification of those components would be changed during software development. Thus an ideal language should be able to allow software

engineers to divide several functions into the requirements of customers and develop different functions each other.

Several functions in an ideal language should easily be merged into the rest of the existing program. If a function can be implemented without consideration of other functions, it cannot always be merged into the rest of the program easily. The errors of an existing program occur by merging several functions into the existing program. Moreover several functions are implemented by different software engineers in parallel. To avoid such errors, a software engineer must thus cooperate with other engineers who implement other functions. And then he must often edit the existing program or the implementation of other functions carefully. For example, name conflict often occurs when several software engineers develop different functions in parallel. When each developer implement his function, the conflict does not occur. However, when functions by several developers is merged into an existing program, it occurs. To avoid name conflict, software developers must cooperate with each other.

Fourth, an ideal language should be a statically typed language. In concurrent development by a certain degree of several developers, the types of modules in a software product should not be changed. It is because those types can be used by several developers as the specifications of the modules. The implementation of a newly appended function often cut across several modules in an existing program. If the types (specifications) of the modules is not changed, a developer who implements the function can safely use the modules. On the other hand, dynamic languages enable changing the type of a module at anytime. To develop a function, a software engineer of its function must carefully use several modules that have already been developed since the type of module may be changed at runtime. If he does not notice that the type is dynamically changed, the instance of the type would cause some errors in the function that he implements. A statically typed language also allows a software engineer to enjoy several support provided by IDEs (Integration Development Environments).

2.3 Example Scenarios

An language suitable for agile software development should satisfy the features shown in Section 2.2. To explain this reason, we illustrate the following two example scenarios on agile software development. These

scenarios are often required by customers on agile software development. The software engineers must rapidly response these needs of the customers and improve the current version of an existing program by the previous iterations.

2.3.1 Scenario 1: Multilingualization of an Online Book Store

For example, by using a method of agile software development, we show that a function for multilingualization is appended to the program for an existing online book store. Recently agile software development has often been used for developing web applications such as an online shopping cart, an online guest book, and so on.

2.3.1.1 Overall architecture of the program of an online book store

Suppose that we were developed a part of the program for the online book store by the previous iterations. A real online book store will be implemented as a collection of components running on top of middleware such as a web application server or a J2EE container. However, the program BookStore of its basic business logic will be something like Figure 2.2. The current version of the book store program is for English.

Clients use their web browser and can enjoy the service provided by the online book store. Once the URL of the book store is accessed by clients, the web application server initializes an object of a servlet representing the web page of the site. A servlet represents the web page required by a client. When an object of a servlet is created, an object of the class BookStore is created as follow.

```
class BookStoreServlet extends HttpServlet {
  BookStore store = new BookStore();
  void doGet(HttpServletRequest request,
             HttpServletResponse response) {
    // creates the web page of the online book store
  }
    ... ...
}
```

To response the GET requests of the clients, the method doGet on the servlet object is invoked by the web application server. Its method creates the HTML document of the book store. The client receives the web server the document.

According to the needs by customers, we were developed a function for keyword search of books that have already been registered in the database. The implementation of the function is a method searchBookTitles. searchBookTitles first invokes a method readDatabase declared in BookStore class and gets an array of book objects of titles associated with the given keyword. Then it outputs the title and price of the each book object.

To use the service of keyword search, a client of a real online book store uses a web browser, types a keyword into a web form, and then sends the book store the keyword via a network. By using reflection, the web application server invokes the method searchBookTitles on the BookStore object through the servlet and passes the received keyword to the method. When the server receives the result of the method, it sends the client the result.

2.3.1.2 The feature for multilingualization

In a current iteration of agile software development, software engineers are appending the feature for multilingualization to the above program of the online book store according to the needs by their customers. Although the current version of the online book store is for english only, the customers require that the online book store enables providing services for several languages. Today, multilingualization of web applications has considerable currency and is thus required by the customers.

To multilingualize the keyword search provided by the current program of the online book store, software engineers would append several buttons to the web page of the keyword search and improve the class BookStore shown in Figure 2.2. Each button responds a service of the keyword search for each language. For example, clients type a keyword into a web form and then push a japanese button to call a searchBookTitles method that was improved for japanese. The improved searchBookTitles method invokes getPrice, getTitle, and getCurrency methods that were also improved for japanese and then returns its result in japanese. The book store displays the clients the result like Figure 2.3.

When developing these features such as Japanese service, French ser-

vice by several software engineers, they divide the function for multilingualization into tasks and then develop different tasks in parallel. In agile software development, they would implement parts of different languages in the implementation of the function. For example, one developer implements a part of Japanese service. Another implements one of French service. Then they would merge their implementations into the current book store when they finished developing all of the parts.

2.3.2 Scenario 2: Visualization of Distributed Software

In the second scenario, we append a function for visualization to an existing raytracing application by a method of agile software development. Raytracing is a general technique from geometrical optics of modeling the path taken by light by following rays of light as they interact with optical surfaces. Its algorithm is often used in computer graphics. By specifing the positions of figures and light, the users of this application can get an image in which the figures were rendered.

2.3.2.1 Overall architecture of a raytracing application

Suppose that we had already developed the calcuration program of raytracing algorithm by the previous iterations. The current version of the program consists of two parts: calcuration programs and a client program. A calcuration program, named Raytracer, is a program that renders (calcurates) the image of the specific figures by raytracing algorithm. Its program runs on two nodes in the current version of the application. The program on each node renders the separate area of the image. On the other hand, a client, named Client, is a program for starting the calcuration program and rendering the specific image (Figure 2.4). Once it is started by a user of this application, a GUI window on which wireframes of figures are drawn is popped up. By moving the wireframes by a mouse, a user can specify the positions of the figures, reflection, refraction and so on that he want to draw.

The code shown in Figure 2.5 is an overview of the Raytracer program. The current version of a raytracing application uses Java RMI which is a framework for distributed object. To start the rendering of an image that a user wants, a client program calls a calcurate method in the raytracer class. Once the calcurate method is given the postion coodinates of the specific figures and light as parameters, it invokes a method

calcurateOnePixel for calcurating pixel data of one dot in the image. The calcurateOnePixel returns a Color object of the position coodinate that was specified as parameters. A Color class is used to encapsulate colors in the RGB color space. The calcurated Color object is serialized and written in a local file by a method writeOnePixel.

To draw an imag, a user must boot the Raytracer programs on two nodes before he starts a client program and uses its GUI window. This reason is because it makes a client program available to access the Raytracer programs on different nodes. A main method in Raytracer makes an instance of Raytracer. And then it registers the instance on a RMI registry that is one of Java standard libraries. The registry runs on the same node as the Raytracer program.

The code shown in Figure 2.6 is an overview of a client named Client. The most part of the client is for displaying a GUI window. A method main first creates an object of it and invokes a method init. The Client object is a main component of a GUI window. The method init set up GUI components and displays its window on a client side. init also calls a method initRaytracers and then initializes a field raytracers in Client. The type of raytracers is an array of the RemoteRaytracer interface shown in Figure 2.5. Client must remotely access two Raytracer programs through the RemoteRaytracer interface. The initRaytracers method first accesses the remote references of RMI registries on remote nodes, gets the remote references of the Raytracer objects in the registries, and set these references to the field raytracers.

A method actionPerformed in Client is the difinition of an event at when pressing a start button. When a user pushes a start button on the GUI window by his mouse, this method is invoked. New threads are created in this method and remotely calls the method calcurate on the Raytracer programs through the RemoteRaytracer interface.

2.3.2.2 The feature for visualization of calcurated data

Here according to the needs of customers, software engineers append the function for visualization of calcurated data to this raytracing application in a current iteration. The function for visualizing the calcurated data is one of important functions since it can help them detect a software bug. As an example, the customers required to add the function for real-time drawing calcurated pixel data on a client side to the current raytracing application like Figure 2.7.

To implement this function, software engineers have to edit two programs: a client program and a calcuration program. They must improve the calcuration program Raytracer since it sends the client side program the pixel data that is rendered by calcurateOnePixel, it has to be improved by the software engineers. Also they must improve the client side program since it receives the passed data from Raytracer and then draws it. Moreover these two programs are different nodes. Thus it is not easy to write the implementation of this function as a module. An ideal language should allow software engineers to write the implementation of one feature as one module.

2.4 Object-Oriented Programming (OOP) Languages

The concept of object-oriented programming (OOP) has been proposed in 1960s. It is a programming paradigm that uses objects and their interactions to design and implement software products. OOP languages provide language constructs for implementing features as modules, named classes. Thus users of these languages can explicitly modularize features. Such OOP languages such as Java, C++, have been getting widely recognized in software industry.

Although OOP langauges allow modularizing features at a certain level, they are not good enough for agile software development. This is because most of these langugaes do not satisfy the above four features of an language suitable for agile software development. Thus existing OOP languages do not allow software engineers to easily add/delete an existing function to/from its software product on agile software development. Also it is not easy for multiple software engineers to develop a software product in parallel. To explain this fact, this section fist illustrates the example programs shown in Section 2.3.1 and 2.3.2 in Java, which is one of existing object-oriented programming languagees.

2.4.1 Solutions in Java

2.4.1.1 Scenario 1

To append multilingualization shown in Section 2.3.1, software engineers would improve the current book store program developed by the previous sections like Figure 2.8. Software engineers first declare methods representing several interfaces that provide multilingualized services for the users. These methods are searchBookTitleForsJapanese, and searchBookTitlesFrench. The users can use a multilingualized book search of the online book store through these methods. A web form for typing a keyword and several buttons appear on the web page. According to the button that the user pushes, he can enjoy the book search in his language. For example, if a user types a keyword into the form and pushes a Japanese button on the web page, the information of the pressed button is notified to the web server. And then the server calls the searchBookTitlesForJapanese method corresponding to the Japanese service.

Several searchBookTitles methods such as searchBookTitlesForJapanese, searchBookTitlesForFrench, invoke a setLang method on a dyContext object and the method searchBookTitles that was developed by the previous iterations. dyContext is declared in the BookStore class as a field. Its type is DynamicContext[2]. The object is created when the BookStore object is initialized. It is used in the searchBookTitles method.

To append this function to the program of book store, the existing method searchBookTitles does not have to be changed. However, the existing methods getPrice, getTitle, and getCurrency must be overridden. According to a language that a user specifies, the method getPrice must switch the calcuration of conversion of dollars into the specified currency. For example, when getPrice is invoked during the execution of the searchBookTitlesForJapanese method, it gets the price of a book given as a parameter, converts the price from dollars into Japanese yen since the return value of getLang is Japanese. And then it returns the converted value. The methods getTitle and getCurrency must also be overridden for adding this function like the getPrice method.

[2]Generally speaking, objects representing dynamic contexts is registered in a session object on a J2EE container. The session object would be one of the libraries provided by the container.

2.4.1.2 Scenario 2

To append the function for visualization to the current raytracing application, software engineers would first improve the Raytracer program on caculation nodes like Figure 2.9. They edit the body of the existing method calcurate. They insert a code for calling a method sendOnePixel just after invoking the method calcurateOnePixel. The method sendOnePixel is newly declared in the class Raytracer by the current iteration. The raytracer program invokes sendOnePixel() and passes its method the pixel data returned by the method calcuratedOnePixel as the third parameter.

By using a remote object callback, the method sendOnePixel sends a client-side program the pixel data given as the parameter. It accesses a rmi registry running on the client-side, tries to get a remote reference of an Callback object that has been registered in advance. If it correctly gets the remote reference, it remotely calls a method drawOnePixel on callback.

Moreover software engineers would improve the client-side programs that had been developed by the previous iteration like Figure 2.10. To draw the pixel data that is received from the Raytracer program, an instance of Callback must be ceated and registered in a rmi registry on a client node before running the calcuration programs. Thus they edit the implementation of the Callback interface and then improve the method main in Client. When the method drawOnePixel is remotely invoked by the raytracer programs, its method uses a field graphics declared in a client view and then renders the received pixel data.

2.4.2 Evaluation

Object-oriented programming (OOP) languages such as Java, C++ are not good enough for agile software development. This is because these languages do not satisfy all the features shown in the previous section.

At first, Java does not satisfy the features (A). When a required feature is written in an OOP language such as Java, a software engineer does not write its implementation without consideration of multiple concerns. In Figure 2.8 of example scenario 2.3.1, to append the feature for multilingualization to original online book store, a software developer must edit the implementations of original getPrice, getTitle, and getCurrency methods in the existing BookStore class. Thus he implements the feature

with consideration of the original book store.

Moreover the new feature in Figure 2.9 and 2.10 of example scenario 2.3.2 cut across the Client and Raytracer programs on different nodes 2.11. A software developer must implement the Callback component on client-side and edit the raytracing program on calcuration node for sending that component pixel data via a network. He must thus implement the feature with consideration of Client, Raytracer, and distribution concerns.

Second, Java does not also satisfy the feature (B). The implementation of a required function often spreads over several parts of the existing program. If the function is removed from the program according to the needs by customers, a software engineer must re-edit the several parts of the existing program. It is not easy task. For example, if customers require the visualization in scenario 2.3.2 in the next iteration, the software engineer must re-edit the Client and Raytracer programs.

Moreover Java does not satisfy the feature (C). Even if several software developers implement different functions, they must often edit same source programs in that software product for the addition of each function. It significantly decreases the productivity of a software program. For example, in example scenario 2.3.1, two developers implement two functions Japanese service and French service that are required by their customers. One developer appends Japanese service to the existing BookStore. Another developer appends French one to the same BookStore. To develop the two functions, they must edit original methods getPrice, getTitle, and getCurrency in the same class BookStore. The time when one developer edits the class BookStore conflicts with the time when another developer edits same one. To avoid this conflict, they must thus cooperate closely with each other and then dicide what order to edit the source program of the class BookStore.

2.4.2.1 Use of delegation pattern

Unlike the ad-hoc implementation of multilingualization in Java, *deletation pattern* allows software engineers to implement this function without editing the existing getPrice, getTitle, and getCurrency methods. According to the needs of their customers, even if they develop services of book search in new languages, they did not have to edit these methods. The program shown in Figure 2.12 is the book store with delegation pattern.

To design the BookStore class with delegation pattern, software engineers implement a class that provides a service for each language. For

example, they declare a class JapaneseService that provides a japanese service of book search (Figure 2.13). JapaneseService implements an interface Service. Once the user of the book search pushes the japanese button on its web page, searchBookTitlesForJapanese is invoked on server-side by reflection. The method searchBookTitlesForJapanese creates an instance of JapaneseService, invokes the method searchBookTitles that was improved in advance. The improved searchBookTitles method newly has a second parameter of the type Service. The seaerchBookTitlesForJapanese method passes the searchBookTitles the instance of JapaneseService.

getPrice, getTitles, and getCurrency that are invoked by searchBookTitles are also improved. These methods newly have a second parameter of the type Service. An instance of the type Service is passed as the second parameter when the each method is called by searchBookTitles. According to the actual type of the instance, the each method delegates the next processing. For example, if the second parameter of getPrice in BookStore is the type JapaneseService, getPrice delegates a method getPrice in JapaneseService. getPrice in JapaneseService converts the price of the given book object from dollar into japanese yen.

Although use of delegation pattern allows software engineers to separate features from other features (satisfing the feature (A)), it does not satisfy the features (B) of an language for agile software development. To develop the application by delegation pattern, the methods getPrice, getTitles, and getCurrency have been refactored before implementing multilingualization. Moreover this pattern does not satisfy the features (C) because software engineers must append searchBookTitlesForJapanese and searchBookTitlesForFrench methods to original BookStore.

2.4.2.2 Using Version Control System

Version control systems such as CVS and subversion are often used when several software engineers develop one software product. A version control system allows developers to implement different versions of its software product in parallel. The Combination Java and CVS enables several developers avoiding to edit a same source code since they append new functions to an existing program in an iteration. Although the combination satisfies a certain degree of the features (C) of a language for agile software development, it is not good enough for agile software development.

For example, two developers append new functions Japanese service

and French service to the existing BookStore program in scenario 2.3.1. To develop the both of Japanese and French services, they must edit the original BookStore each other. CVS allows the software developers to implement different functions without editing the existing BookStore program. To implement Japanese service, one developer edits classes BookStore1 that is different version of the existing BookStore classe. To implement French service, another developer also edits classe Bookstore2 that is these different version of the original class. To append two functions to the existing BookStore program, they would merge classes BookStore1 and BookStore2 into the current class BookStore.

However, it is not easy to merge different versions of a class by existing version control systems. To merge these different versions, developers must carefully review each source code and then check if the code conflicts or not. For example, the two software developers improve methods getPrice, getTitle, and getCurrency in each version of original BookStore. To merge classes BookStore1 and BookStore2 into BookStore, they must change the implementations of the original getPrice, getTitle, and getCurrency by hand. This does not satisfy the feature (B) of a language for agile software development.

2.4.2.3 Dynamic Scoping

Dynamic languages that have dynamic scoping such as Common Lisp satisfies the feature (C) of a language for agile software development. For example, dynamic scoping in Common Lisp allows switching service objects such as Japanese service, French service that are appended to the BookStore according to control flow. Also since Common Lisp provides a mechanism for a hook, it satisfies the feature (A). Japanese and French services can be separated from the rest of the current program.

However, dynamic languages do not satisfy the feature (D). In concurrent development by a certain degree of several developers, the types of modules in a software product should not be changed. It is because those types can be used by several developers as the specifications of the modules. The implementation of a newly appended function often cut across several modules in an existing program. If the types (specifications) of the modules is not changed, a developer who implements the function can safely use the modules. On the other hand, dynamic languages such as Common Lisp and Perl enable changing the type of a module at anytime. To develop a function, a software engineer of its

function must carefully use several modules that have already been developed since the type of module may be changed at runtime. If he does not notice that the type is dynamically changed, the instance of the type would cause some errors in the function that he implements.

2.5 Inheritance

In object-oriented programming, inheritance is methodology to form new classes using classes that have been defined already. The new version's classes, known as derived classes, take over (or inherit) attributes and behavior of the pre-existing classes, which are referred to as base classes (or ancestor classes). It is intended to help reuse existing code with little or no modification.

2.5.1 Various Inheritance

2.5.1.1 Alternative inheritance

To extend an original class and an original class hierarchy, many number of researchers have proposed and developed extensions to the idea of inheritance such as mixins (or mixin layers) [70, 22], traits [52], MixJuice [83], MJ [36], virtual classes [47], nested inheritance [55] and so on. However, these technologies do not satisfy three requirements shown in the previous section.

Cook presents a use of inheritance as a derivation of modified hierarchies or other graph structures [26]. Links between nodes in a graph are interpreted as self-references from within the graph to itself. By inheriting the graph and modifying individual nodes, any access to the original nodes is redirected to the modified versions. For example, a complete class hierarchy may be inherited, while new definitions are derived for some internal classes. The result of this inheritance is a modified class hierarchy with the same basic structure as the original, but in which the behavior of all classes modified that depend upon the classes explicitly changed is modified. Hierarchy inheritance is based on having a lookup of classes and on relationship between group of classes, whereas with classboxes, no class-lookup is involved and import is done at the class-level.

Virtual classes [47] are a concept from BETA [48] (known as virtual pattern in BETA) and have been supported by several languages such as Caesar [50] and Keris [49]. Unlike Java, BETA allows extending not only methods, fields but member classes in the classes. Member classes that are included in its subclass can override the original member classes in the superclass. Moreover, gbeta provides a family polymorphism, which is a function for extending virtual class hierarchies. An sub-hierarchy can reuse the methods in other sub-class hierarchies. Those functions allow appending new methods that are declared in other sub-hierarchies.

The Jx programming language is an extension of Java where members of an encapsulating class or package may be enhanced in a subclass or subpackage [55]. Packages may have a declared inheritance relationship. Nested classes in Jx are similar to virtual classes. Unlike virtual classes, nested classes in Jx are attributes to their enclosing class, not attributes of instances of their enclosing class.

The Scala programming language is a statically-typed object-oriented and functional programming language developed at EPFL. It contains a construct called *views*, which are a form of language support for adaptation [12]. Views are essentially programmer-defined functions from one type to another. However, the language infers where a view should be inserted, in order to allow a value of one type to be treated as having a different type. Views must be explicitly imported in order to be considered for such inference. On the implementation side views provide no special support for adaptation. For example, in order to augment a type with new methods, the developers have to implement a view that explicitly creates a wrapper class for values of that type, along with forwarding methods for all of the original methods of that type.

2.5.1.2 Mixin Layer

Mixins are classes that provide a functionality to be inherited by a subclass, but is not meant to standalone. Inheriting from mixin is not a form of specialization but is rather a means to collect functionality. A subclass may even choose to inherit most or all of its functionality by inheriting from one or more mixins through multiple inheritance.

A mixin can also be viewed as an interface with implemented methods. When a class includes a mixin, the class implements the interface and includes, not inherits, all the mixin's attributes and methods. They become part of the class during compilation. Interestingly enough mixins

don't need to implement an interface. The advantage of implementing an interface is obvious so that the class may be passed to methods requiring that interface.

2.5.2 Solutions with Java Inheritance

Normal Java language provides a language construct for single inheritance. A class enables inheriting behaviors defined in another class, named a superclass. Java inheritance allows us developing the features (multilingualization and visualization) in the above scenarios as follow.

2.5.2.1 Scenario 1

By using Java inheritance, software engineers can separate the program that they want to implement from the rest of the program to a certain degree. Moreover its program can be developed without consideration of other newly appended features and the rest of the program to a certain degree.

For example, the program for Japanese book search with Java inheritance does not include other features of multilingualization shown in Figure 2.14. The program is named JapaneseBookStore. It extends the existing class BookStore. It has new methods and field searchBookTitlesForEnglish, searchBookTitlesForJapanese, and dyContext. And then it overrides three methods getPrice, getTitle, and getCurrency that were defined in existing BookStore. See the method getPrice in the class JapaneseBookStore. The methodc converts the price of the given book object from dollar into Japanese yen if the return value of getLang is Japanese. Otherwise, It invokes getPrice in original BookStore. Also the methods getTitle and getCurrency include the concern for Japanese book search only.

To append the function for French book search to the current program, software engineers would declare a class FrenchAndJapaneseBookStore that extends the class JapaneseBookStore like Figure 2.15. The class FrenchAndJapaneseBookStore has new methods searchBookTitlesForFrench and overriding methods getPrice, getTitle, and getCurrency. The method getPrice converts the price of the given book object from dollar to euro if the return value of getLang is French. Otherwise, it invokes original getPrice. getTitle and getCurrency are also overridden.

Moreover to replace the current book store program with the one of the multilingualized book store, software engineers must the servlet program representing a web page of book store like the following.

```
class BookStoreServlet extends HttpServlet {
  FrenchAndJapaneseBookStore store
    = new FrenchAndJapaneseBookStore();
  void doGet(HttpServletRequest request,
             HttpServletResponse response) {
    // creates the web page of the online book store
  }
    ... ...
}
```

2.5.2.2 Scenario 2

By using Java inheritance, software engineers can separate the implementation for virtualization required by their customers from the rest of the program to a certain degree. See Figure 2.16. The programs in the Figure are written with inheritance. A RaytracerWithCallback class extends original Raytracer class. A calcurateOnePixel method in the class RaytracerWithCallback overrides original one. When the overridden method calcurateOnePixel is invoked, the overridden one calls original one. Then it invokes a method sendOnePixel. The sendOnePixel method is newly appended to the RaytracerWithCallback class and sends the callback component the pixel data returned by original calcurateOnePixel via a network.

2.5.3 Evaluation

Although inheritance allows separating Japanese and French service from the original BookStore class in example scenario 2.3.1, it is not good enough for agile software development. It does not satisfies the feature (A) too much. This is because the feature for visualization consists of two distributed sub-components on client node and calcuration node. This feature cannot be implemented without consideration of distribution concern.

Software developers can write the implementations of new functions as subclasses however, they must change client programs of original classes into a code to call these subclasses for appending the functions to

the program. Thus inheritance does not satisfy the feature (B). For example, to implement the feature for multilingualization, software developers write subclasses JapaneseBookStore and FrenchAndJapaneseBookStore of original BookStore. To append the feature to the current book store program, they must check all of the existing book store program and then replace client codes for creating objects of BookStore with ones for creating objects of FrenchAndJapaneseBookStore.

In example scenario 2.3.1, to add multilingualization to the current program, the code who software developers replace is only one in the class BookStoreServlet. However, generally speaking, there are several client codes in a program. Thus the replacement is not a simple task. Also some readers would think that use of *factory pattern* allows reducing the number of such client codes. However, software engineers would not design a program in which all instances are created with factory pattern.

Moreover inheritance does not satisfy the feature (C). It allows separating Japanese and French services from the program of online book store. However, French service depends on Japanese service. For example, the class FrenchAndJapaneseBookStore extends the class JapaneseBookStore. Then FrenchAndJapaneseBookStore uses the field dyContext declared in JapaneseBookStore. Thus French service cannot be implemented without consideration of Japanese service.

2.6 Aspect-Oriented Programming (AOP) Languages

In 1997 Aspect-Oriented Programming (AOP) has been proposed as technology for improving separation of concerns within software by Kiczales [33]. However Object-Oriented Programming (OOP) is a technology that can modularize the source codes of software systems as class modules, it is not sufficient technology enough to separate concerns that are scattered throughout several modules. Such scattered concerns are called *crosscutting concerns*.

Crosscutting concerns decrease maintainability and understandability of software systems. To understand a crosscutting concern, developers must read the source codes of all modules that the concern cut accross. Also to edit the implementation of a crosscutting concern, developers must edit multiple modules. For example, codes for logging are scattered

within code whose primary responsibility is something else. Even if the developers change and remove the codes for logging, they must change and remove every scattered codes for logging.

AOP can pull the widespread crosscutting concern into a single module. These modules are termed *aspects*. AOP builds on several technologies, such as procedural programming and OOP, that have already made significant improvements in software modularity.

2.6.1 Various AOP Languages

Several general-purpose AOP languages, tools, and systems have been proposed and developed before. AspectJ [38] is a simple and practical AOP based extension to Java. Adaptive Programming provides a special-purpose language, called DemeterJ [58], for writing class structure traversal specifications. DemeterJ prevents knowledge of the complete class structure from becoming tangled throughout the code. Composition filters object model [20] provides control over messages received and sent by an object. The composition filters mechanism provides an aspect language that can beused to control a number of aspects inlcuding synchronization and communication. ComposeJ [81] is an extension of the Java language that adds composition filters to Java classes through inlining. Multi-dimensional separation of concerns (Subject-Oriented Programming) [76] provides for composing and integration disparate class hierarchies, each of which might represent different concerns. Hyper/J [59] supports separation and integration of concerns along multi-dimensional in standard Java software. JAC [62, 60, 61, 57] is a Java framework for dynamic AOP. Unlike other lanugages such as AspectJ, JAC does not require any langugae extensions to Java.

2.6.1.1 AspectJ

AspectJ is an implementation of AOP for Java [38, 2] [3]. But concerns, such as logging and security, cut across the classes in Java, the crosscutting concerns are not easily turned into classes precisely. AspectJ allows developers to implement this concerns in Java.

AspectJ adds to Java just one new concept, a *join point* – and that's really just a name for an existing Java concept. It adds to Java only a

[3]In this paper, we call AspectJ 1.0.6 "AspectJ". Currently, AspectJ 1.2.1 have been released.

few new constructs: *pointcuts*, *advice*, *inter-type declarations* and *aspects*. Pointcuts and advice dynamically affect a thread of control, inter-type declarations statically affects a program's class heirarchy, and aspects encapsulate these new constructs.

Join Points A *join point* is a program's operation in the program flow. As an example of operations, there are method calls, method executions, field accesses, constructor calls, constructor executions, and exception events in the program flow. A join point model of many AOP languages, frameworks, and systems are based on AspectJ.

Pointcuts *Pointcuts* pick out certain join points in the program flow. For example, the pointcut:

```
call(void Point.setX(int))
```

picks out join points that is a method call of the signature void Point.setX(int) in the program flow. The call is one of pointcut designators, identifies each join points that are call of the specified method. In table 2.1, we listed several pointcut designators that AspectJ is provided.

Table 2.1. Several pointcut designators of AspectJ

designator	join points
within(*TypePattern)*	the join points included in the declaration of the types matching *TypePattern*
target(*Type or Id*)	the join points where the target object is an instance of *Type* or the type of *Id*
this(*Types or Ids*)	the join points when the currently executing object is an instance of *Type* or *Id*'s type
args(*Types or Ids*)	the join points where the arguments are instances of *Types* or the types of of the *Ids*
call(*Signature*)	the calls to the methods matching *Signature*
execution(*Signature*)	the execution of the methods matching *Signature*
cflow(*Pointcut*)	all join points that occur between the entry and exit of each join point specified by *Pointcut*

Also, a pointcut can be built out of other pointcuts with and (&&), or (||), and not (!). For example, the pointcut:

```
call(void Point.setX(int)) || call(void Point.setY(int))
```

picks out join points that is method call of the signature void Point.setX(int) or void Point.setY(int).

Advice *Advice* is a method-like mechanizm used to declare that certain code should execute at each join point in a pointcut. Advice consists of two parts: the one is a pointcut, the other is the code. AspectJ has before, after and around advice. before advice runs before the thread of control reaches each join point in a pointcut. after advice runs after the thread of control reaches identified join points.

Aspects *Aspects* are modular units of crosscutting implementation, wrap up pointcuts and advice. It is defined like a class, can have member methods and fields. As an example, the aspect:

```
aspect LoggingAspect {
   pointcut move():
      call(void Point.setX(int)
        || void Point.setY(int));
   before(): move() {
      System.out.println("about to move");
   }
}
```

prints a message whenever the setX() and setY() method in the Point class are called.

Inter-type Declarations *Inter-type declarations* (formerly called the *introduction*) are declarations that cut across classes and their hierarchies in AspectJ. We can declare those in an aspect.

Recently, AspectJ's development team provides AJDT, which is an eclipse plugin for AspectJ. By using AJDT, developers can write the programs of AspectJ on the customized Eclipse IDE and users can enjoy the code assist supported by the eclipse.

2.6.1.2 CaesarJ

CaesarJ [50, 16, 4] is an AOP language providing an expressive power and is an extension to the standard Java syntax. CaesarJ takes a step further

and ensures other important properties of modularity: abstraction, information hiding and minimization of dependencies. Aspects are designed as components, which have clear abstraction and can be reusable. In order to achieve these goals CaesarJ slightly extends run-time conception of object-oriented systems by grouping objects to collaborations, using virtual types and bindings.

CaesarJ improves separation of concern in the same way as AspectJ. AspectJ style pointcuts and advices can be used to intercept points, where component functionality should be integrated. CaesarJ also modularizes components, which consist of multiple collaborating classes. Developer can define collaboration interface, which is a set of related Java interfaces required and provided by the component. Then they implement the component in using collaboration interface. In a separate module they define the binding of the component to the application. They achieve information hiding, because component implementation is abstracted from application specific concepts, and on the other hand component implementation details are hidden behind the collaboration interface. Moreover CaesarJ enables modularization of different features within a class collaboration. By using virtual classes, developer can define base class collaboration and incrementally refine it with new features in its subcollaborations. Independently developed features can be smoothly merged by applying mix-in composition on the subcollaborations.

CaesarJ also provides an Eclipse plugin, named Caesar plugin, for customizing the normal Eclipse IDE same as AJDT. The plugin enables those developers to develop the program of CaesarJ on the customized Eclipse. The developers can exploit coding support such as a code assist provided by the customized Eclipse.

2.6.1.3 Annotation based AOP frameworks

AspectJ5 [11], AspectWerkz [3], and JBoss AOP [6], which are annotation-based AOP frameworks, allow programmers to write aspects in the normal Java syntax. AspectJ5, AspectWerkz, and JBoss AOP have an expressive power that is equivarent with one of pointcut-advice provided by AspectJ's. The contribution of their frameworks enable developers to write the programs on any Java IDE without customizing the IDE. To use the frameworks, developers do not need to change the their favorite IDE into a specific IDE.

2.6.1.4 Dynamic AOP

PROSE [64, 63] is efficient dynamic AOP system. and allows dynamically weaving and unweaving aspects including pointcut-advices. The implementation of PROSE employs the Java Platform Debugger Architecture (JPDA) [72]. JPDA can capture relevant execution events and intercept the program execution. PROSE checks whether or not an intercepted event is specified as a pointcut. If the event is specified, it executes the corresponding advice.

JAsCo [75], AspectWerkz [3], and Wool [69] are dynamic AOP frameworks based on the Java HotSwap [30]. Java HotSwap allows dynamically reloading a class definition. However, a newly loaded class definition can only changed the implementation of a method declared in its original definition. It cannot add or remove a method or a field to the original class. These frameworks allows not only weaving pointcut-advices but overriding methods in an original class during runtime.

Steamloom [21] is a powerful dynamic AOP system and provides a customized Java virtual machine (IBM's Jikes RVM [7]) for users. The current version of Steamloom supports a mechanism for dynamically weaving only pointcut-advices. Sister namespaces [68] also extend Java virtual machine (IBM's Jikes RVM) and allow dynamically exchanging instances of different versions of a class. If there is an AOP system based on sister namespaces, the system allows weaving and unweaving aspects that include not only pointcut-advices but also intertype declarations during runtime.

These languages, frameworks, and systems (PROSE, JAsCo, AspectWerkz, Wool, and Steamloom) support a mechaism enough for dynamically modularizing crosscutting concerns that users cannot anticipate in advance at development time. Their importance has been getting widely recognized. However, the applications of dynamic AOP is not limited to that case. It is often convenient to be able to dynamically switch aspects even if they know all the aspects in advance.

Wasif et al. proposed family-based dynamic weaver framework [80]. In that paper, they also proposed the concept of dynamic AOP with static preparation. The concept is of an aspect weaver that statically makes the preparation for dynamically switching aspects if all the aspects are known at development time. It is similar to Dynamic GluonJ's. However, a dynamic weaver for it has not been supported and implemented in their framework. Our paper presented the specification and implementation

of language constructs for the concept.

2.6.1.5 Refinement

Refinements allow developers to write an extension to the definition of an original class. Unlike inheritance and mixin mechanisms, however, refinements directly modify the original class definition. Thus to use the extended class definition, its clients do not need to explicitly make a new instance of the extended class.

In 2003 *A classbox* is one of concepts for extending classes and was originally developed with Smalltalk [14]. The extension of Java for supporting the concept of classboxes, called Classbox/J [19], were recently implemented. Classbox/J allows not only appending new members (*i.e.* fields, methods, constructors, and so on) to existing classes but overriding original methods in those classes. The refined classes can be explicitly imported from other classes. Classbox/J can dynamically collaborate to control the scope of an extension of an existing class. In other words, it can switch implementations of methods according to caller-side programs that invoke those methods.

eJava [79] is the extension of Java and provides language constructs, called *expanders*. Expanders allow appending new fields, methods, and interfaces to an original class. To call new methods appended by expanders, its programmers import the expanders into caller-side programs. When the programmers append a feature to a class hierarchy that consists of several classes, they can describe the feature that is appended to each original class as an expander. Then the expanders can be grouped as an expander family. The function has powerful expressiveness. All expanders statically extend original classes. In other words, eJava does not provide language constructs for dynamically changing an original class. It cannot change the behavior of an object of its extended class during runtime.

Lieberman originally introduced delegation in the framework of a prototype-based object model [45]. An object (child) may have references to other objects (parents). If a method that a child does not have is executed, the extension is automatically forwarded to a method with the same name on its parents. Darwin/Lava [42] moreover supports dynamic delegation for Java. Dynamic delegation allows changing the behavior of a child object during runtime by using changing its parent objects. In other words, it enables switching a different set of method

implementations on the child object even though its object have been already created. We can say the same for delegation layers [40, 41], which are functions for integrating virtual classes and delegation.

2.6.2 Solutions in AspectJ

2.6.2.1 Scenario 1

The aspects JapaneseService and FrenchService in Figure 2.17 are the implementation of the feature for multilingualization in scenario 2.3.1. These aspects include each three around advices and one inter-type declaration.

For example, the first advice in JapaneseService modifies the behavior of the getPrice method in the original BookStore class during execution of a method searchBookTitlesForJapanese. The searchBookTitlesForJapanese method is appended to the original BookStore class by using an inter-type declaration. The second and third advices also modify the behavior of the getTitle and getCurrency methods during the same period.

On the other hand, the first advice in FrenchService modifies the behavior of the getPrice method in original BookStore during execution of a method searchBookTitlesForFrench, which is appended to BookStore by using an inter-type declaration. The second and third advices modify the behavior of the getTitle and getCurrency methods during the period.

2.6.2.2 Scenario 2

AspectJ allows writing the visualization code without editing the current raytracing application like Figure 2.18. The aspect PixelDataGetting is allocated on calcuration node. It specifies a call of a method calcurateOnePixel as a pointcut. Then in its advice, the callback component on client node is invoked. Moreover a software developer must implement the Callback component on client node before starting the application. This is because the advice invokes the component.

2.6.3 Evaluation

First, AspectJ satisfies the feature (B) and (D). AspectJ is an extension to Java, which is a statically typed language. Then the programs in

Figure 2.17 and 2.18 is written without editing the original programs BookStore, Client, and Raytracer.

Next, AspectJ does not satisfies the feature (A) too much. Although AspectJ allows separating the feature for visualization can be separated from the rest of the reytracing application, the implementation consists of two distributed sub-components like Figure 2.19. AspectJ allows separating the code for getting the calcurated pixel data from the existing Raytracer program as an aspect, named PixelDataGetting. The aspect remotely sends the calcurated data to the Callback program when it gets the data. The Callback component draws the data on the client's window.

As we can see, even this visualization is implemented by distributed sub-components and hence we had to write complicated network processing code using Java RMI despite that it is not related to the visualization concern. In particular, the PixelDataGetting aspect is used only for getting the calcurated data by calcurateOnePixel on each calcuration node. The PixelDataGetting aspect is a sub-component that is necessary only because calcurateOnePixel() and the client's window are deployed on different nodes. This means that the component design of the visualization is influenced by concerns about distributed. Furthermore, this aspect is similar to what the AspectJ compiler produces for implementing the pointcut-advice framework. It should not be hand-coded, but implicit within the language constructs provided by an AOP language.

Moreover, AspectJ does not satisfy the feature (C) too much. Although cflow pointcut in AspectJ allows several engineers to develop different features as aspects, the implementation of each aspect are not simple. The cflow expressions specified for each advice would be unnecessary and redundant. Appending a cflow pointcut expression to each advice will be tedious if the number of advices is large.

2.7 Other Approaches

Hyper/J [59] is a subject-oriented programming (SOP) language for Java. Hyper/J is based on the notion of *hyperspaces*, and promotes composition of independent concerns at different times. *Hyperslices* are building blocks containing fragments of class definitions. They are intended to be composed to form larger building blocks called *hypermodules*. A hyperslice defines methods for classes that are not necessarily defined in that hyperslice: class members are spread over several hyperslices. Hyper/J

can statically divide the existing program into several parts and then merge the divided parts and other programs. Although language constructs provided by Hyper/J are powerful, Hyper/J needs to transform a program statically. The program transformed by Hyper/J cannot be changed during runtime.

MultiJava [29] is an extension of Java for providing open classes and multiple method dispatch. Its language construct is called *open classes*. It allows appending new methods to existing classes statically. To invoke appended methods, users of MultiJava explicitly import those methods into caller-side programs. However, MultiJava does not support appending new fields or overriding existing methods. It moreover extends existing classes statically.

In 2005 context-based programming is a concept for dynamically extending original classes and enables switching the extension of original classes according to dynamic contexts during runtime. It was originally developed with CLOS [27]. ContextJ [28] is the extension of Java for the concept. It allows programmers to override an existing method in its original class and to change multiple overriding methods according to dynamic contexts. However, it does not allow appending a new method to an original class according to dynamic contexts. Thus it cannot change the behavior of an object of its original class during runtime.

Half and Half is an extension to Java that supports the ability to add new superinterfaces to existing classes, as well as a form of multiple dispatch. A wrapper strategy is similar to the eJava and is used to compile the new language construct. Half and Half does not support the addition of new methods or fields to existing classes. Therefore, a new interface can only be given to an existing class if it already meets all the requirements of that interface.

2.8 Summary

This chapter explained the motivation of our work. Although some of existing programming languages have been used by software engineers for agile software development, these languages are not good enough for agile software development. Then we presented the four features of a language for agile software development. These features were as follow.

(A) Separation of features

(B) Without editing an existing program

(C) Concurrent development

(D) A statically typed language

Unfortunately, existing programming languages do not satisfy all the four features (Table 2.2). Most of existing languages do not satisfy the feature (A) too much. This is because these languages do not enable separating visualization from the raytracing application even if these ones allows separating multilingualization from the book store. These languages does not provide mechanisms or language constructs for distributed features like visualization.

In the table, AspectJ is the most suitable programming language for agile software development in existing programming languages and techniques. However, AspectJ is not good enough for agile software development. According to the Table 2.2, although AspectJ satisfies features (B) and (D), it does not satisfy features (A) and (C).

Although AspectJ allows separating the function for visualization from the raytracing application as a component, the implementation of its component is not simple and consists of several distributed subcomponents. To communicate these sub-components via a network, the programs of these sub-components include complicated network processing codes like Java RMI. AspectJ cannot modularize such functions for distributed software without consideration of distribution. An ideal language should allows separating these functions as simple aspects, which are non-distributed modules.

Although cflow pointcut in AspectJ enables multiple software engineers to develop different features at same iteration, these aspect implementations are not simple. The cflow expressions specified for each advice would be unnecessary and redundant. Appending a cflow pointcut expression to each advice will be tedious if the number of advices is large.

The target application in this thesis is development of web applications such as the program of an online book store shown in Section 2.3.1. The programs of most web applications are particular forms. These programs have session objects, provide a way to identify a user access, and a mechanism for managing those objects. Dynamic contexts of a web application are registered and centralized in a session object. A software engineer of a web application uses a session object and describe

the codes for switching processings according to session objects by hand. We should use sessions and a session management mechanism. Then, for concurrent development, we should propose a language construct that allows multiple software engineers to simply develop different processings according to the session objects.

```
class BookStore {
  String searchBookTitles(String keyword) {
    Book[] books = readDatabase(keyword);
    StringBuffer sbuf = new StringBuffer();
    for (Book b : books) {
      sbuf.append(getTitle(b))
          .append(",")
          .append(getCurrency())
          .append(getPrice(b))
          .append("\n");
    }
    return sbuf.toString();
  }
  Book[] readDatabase(String keyword) {
    // accesss a database and makes book objects of
    // titls associated with the given keyword.
  }
  int getPrice(Book b) {
    return b.price();
  }
  String getTitle(Book b) {
    return b.title();
  }
  String getCurrency() {
    return "USD";
  }
    ... ...
}
```

Figure 2.2. The current program of an online book store in Java

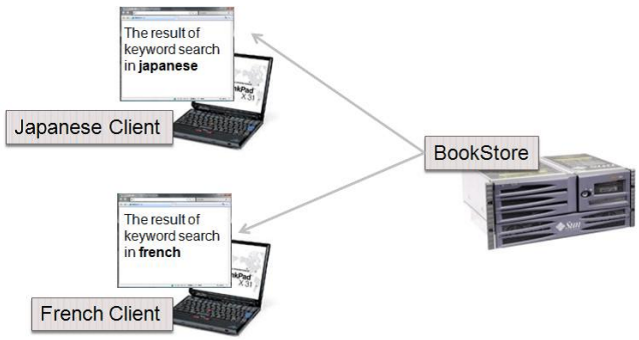

Figure 2.3. Architecture of the program of online book store

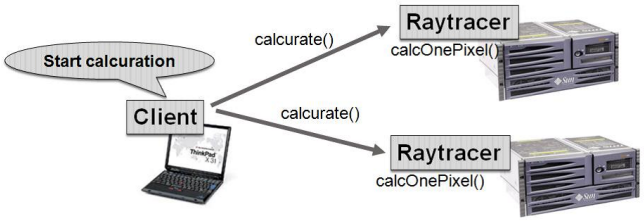

Figure 2.4. Architecture of the program of a raytracing application

```
class Raytracer implements RemoteRaytracer {
  void calcurate(List figures, Light light) {
    for (int ypos = 0; ypos < IMAGE_SIZE; ypos++) {
      for (int xpos = 0; xpos < xlen; xpos++) {
        Color c = calcurateOnePixel(xpos, ypos);
        writeOnePixel(xpos, ypos, c);
      }
    }
  }
  Color calcurateOnePixel(int x, int y) {
    // Calcurates pixel data of one dot that was
    // specified as parameters and returns its result
  }
  static void main(String[] args) {
    RemoteRaytracer raytracer = new Raytracer();
    Registry registry =
      LocateRegistry.createRegistry(1099);
    registry.bind("raytracer", raytracer);
  }
    ... ...
}
interface RemoteRaytracer extends Remote {
  void calcurate(List figures, Light light);
}
```

Figure 2.5. An overview of a Raytracer program in Java

```
class Client extends JFrame {
  String[] nodes;
  RemoteRaytracer[] raytracers;
  static Graphics graphics;
  void init() {
    // Set up a GUI window
    // and initializes a field named raytracers
  }
  void initRaytracers() {
    nodes = new String[] { "node1", "node2" };
    raytracers = new RemoteRaytracer[2];
    for (int i = 0; i < raytracers.length; i++) {
      Registry registry =
        Registry.getRegistry(nodes[i], 1099);
      raytracers[i] = registry.lookup("raytracer");
    }
  }
  void actionPerformed(ActionEvent event) {
    if (event.getSource() == startButton) {
      for (int i = 0; i < raytracers.length; i++) {
        new Thread() {
          void run() {
            raytracers[i].calcurate(figures, light);
          }
        }.start();
      }
    }
  }
  static void main(String[] args) {
    Client client = new Client();
    client.init();
  }
    ... ...
}
```

Figure 2.6. An overview of the Client program in Java

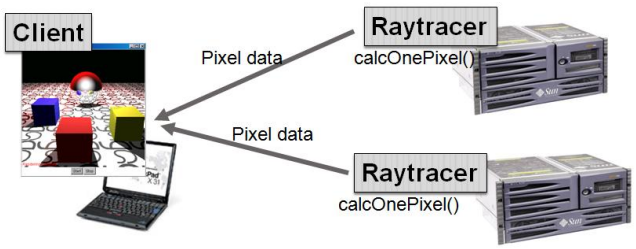

Figure 2.7. Architecture of a feature for real-time drawing

```
class BookStore {
  DynamicContext dyContext;
  String searchBookTitlesForJapanese(String kw) {
    dyContext.setLang("JPN");
    return searchBookTitles(kw);
  }
  String searchBookTitlesForFrench(String kw) {
    dyContext.setLang("FRN");
    return searchBookTitles(kw);
  }
  String searchBookTitles(String keyword) {
    // this method was developed by the previous
    // iterations
  }
  int getPrice(Book b) {
    if (dyContext.getLang().equals("JPN"))
      return b.price() * 105;
    } else if (dyContext.getLang().equals("FRN") {
      return b.price() * 7 / 10;
    } else { // for english
      return b.price();
    }
  }
  String getTitle(Book b) {
    // This method is also overridden like getPrice.
    // According to the return value of getLang, it
    // switches the notation of the given book title.
  }
  String getCurrency() {
    // This method is also overridden like getPrice.
  }
    ... ...
}
```

Figure 2.8. The implementation of the function for multilingualization in Java

```
class Raytracer implements RemoteRaytracer {
  Callback callback;
  void calcurate(List figures, Light light) {
    for (int ypos = 0; ypos < IMAGE_SIZE; ypos++) {
      for (int xpos = 0; xpos < xlen; xpos++) {
        Color c = calcurateOnePixel(xpos, ypos);
        sendOnePixel(xpos, ypos, c);
        writeOnePixel(xpos, ypos, c);
      }
    }
  }
  void sendOnePixel(int x, int y, Color c) {
    if (callback == null) {
      Registry registry =
        getRegistry("client", 1099);
      callback = (Callback)
        registry.lookup("callback");
    }
    callback.drawOnePixel(xpos, ypos, c);
  }
    ... ...
}
```

Figure 2.9. An overview of the Raytracer program improved for visualization in Java

```
class Client extends JFrame {
  class CallbackImpl implements Callback {
    void drawOnePixel(int x, int y, Color c) {
      synchronized(graphics) {
        graphics.setColor(c);
        graphics.drawLine(x, y, x, y);
      }
    }
  }
  static void main(String[] args) {
    Callback callback = new CallbackImpl();
    Registry registry =
      LocateRegistry.createRegistry(1099);
    registry.bind("callback", callback);
    ... ...
  }
    ... ...
}
interface Callback {
  void drawOnePixel(int x, int y, Color c);
}
```

Figure 2.10. An overview of the client-side program for visualization in Java

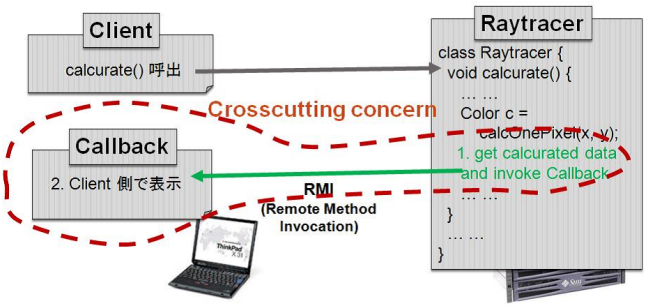

Figure 2.11. A feature for real-time drawing cut across client-side and server-side programs

```
class BookStore {
  String searchBookTitlesForJapanese(String kw) {
    Service serv = new JapaneseService();
    return searchBookTitles(kw, serv);
  }
  String searchBookTitlesForFrench(String kw) {
    Service serv = new FrenchService();
    return searchBookTitles(kw, serv);
  }
  String searchBookTitles(String keyword, Service serv) {
    Book[] books = readDatabase(keyword);
    StringBuffer sbuf = new StringBuffer();
    for (Book b : books) {
      sbuf.append(getTitle(b, serv))
          .append(",")
          .append(getCurrency(serv))
          .append(getPrice(b, serv))
          .append("\n");
    }
    return sbuf.toString();
  }
  int getPrice(Book b, Service serv) {
    if (serv == null) return b.price();
    return serv.getPrice(b);
  }
  String getTitle(Book b, Service serv) {
    if (serv == null) return b.title();
    return serv.getTitle(b);
  }
  String getCurrency(Service serv) {
    ... ...
  }
    ... ...
}
```

Figure 2.12. The implementation of multilingualization using delegation pattern in Java

```
interface Service {
  int getPrice(Book b);
  String getTitle(Book b);
  String getCurrency();
}
class JapaneseService implements Service {
  int getPrice(Book b) {
    return b.price() * 105;
  }
  String getTitle(Book b) {
    // ... ...
  }
  String getCurrency() {
    // ... ...
  }
}
```

Figure 2.13. The implementations of JapaneseService and Service using delegation pattern in Java

```
class JapaneseBookStore extends BookStore {
  DynamicContext dyContext = new DynamicContext();
  String searchBookTitlesForJapanese(String kw) {
    dyContext.setLang("JPN");
    return searchBookTitles(kw);
  }
  String searchBookTitles(String keyword) {
    // this method was developed by the previous
    // iterations
  }
  int getPrice(Book b) {
    if (dyContext.getLang().equals("JPN"))
      return b.price() * 105;
    } else {
      return super.getPrice(b);
    }
  }
  String getTitle(Book b) {
    // overrides this method for Japanese service
  }
  String getCurrency() {
    // overrides this method for Japanese service
  }
    ... ...
}
```

Figure 2.14. An extension to BookStore for Japanese book search with Java inheritance

```
class FrenchAndJapaneseBookStore
                    extends JapaneseBookStore {
  String searchBookTitlesForFrench(String kw) {
    dyContext.setLang("FRN");
    return searchBookTitles(kw);
  }
  int getPrice(Book b) {
    if (dyContext.getLang().equals("FRN"))
      return b.price() * 7 / 10;
    } else {
      return super.getPrice(b);
    }
  }
  String getTitle(Book b) {
    //
  }
  String getCurrency() {
    //
  }
    ... ...
}
```

Figure 2.15. An extension to BookStore for French book search with Java inheritance

```
interface Callback {
  void drawOnePixel(int x, int y, Color c);
}
class RaytracerWithCallback extends Raytracer
    implements RemoteRaytracer {
  Callback callback;
  Color calcurateOnePixel(int x, int y) {
    Color c = super.calcurateOnePixel(x, y);
    sendOnePixel(x, y, c);
    return c;
  }
  void sendOnePixel(int x, int y, Color c) {
    if (callback == null) {
      Registry registry =
        getRegistry("client", 1099);
      callback = (Callback)
        registry.lookup("callback");
    }
    callback.drawOnePixel(xpos, ypos, c);
  }
    ... ...
}
```

Figure 2.16. An extension to the existing raytracing program for visualization with Java inheritance

```
aspect JapaneseService {
  String BookStore.searchBookTitlesForJapanese(String kw) {
    return searchBookTitles(kw);
  }
  int around():
    cflow(String BookStore.searchBookTitlesForJapanese(..))
    && args() && execution(int BookStore.getPrice(Book)) {
    return proceed() * 105;
  }
  int around():
    cflow(String BookStore.searchBookTitlesForJapanese(..))
    && args() && execution(String BookStore.getTitle(Book)) {
    // an extension to getTitle for Japanese service
  }
  int around():
    cflow(String BookStore.searchBookTitlesForJapanese(..))
    && args() && execution(String BookStore.getCurrency(Book)) {
    // an extension to getCurrency for Japanese service
  }
}
aspect FrenchService {
  String BookStore.searchBookTitlesForFrench(String kw) {
    return searchBookTitles(kw);
  }
  int around():
    cflow(String BookStore.searchBookTitlesForFrench(..))
    && args() && execution(int BookStore.getPrice(Book)) {
    return proceed() * 7 / 10;
  }
  int around():
    cflow(String BookStore.searchBookTitlesForFrench(..))
    && args() && execution(String BookStore.getTitle(Book)) {
    // an extension to getTitle for French service
  }
  int around():
    cflow(String BookStore.searchBookTitlesForFrench(..))
    && args() && execution(String BookStore.getCurrency(Book)) {
    // an extension to getCurrency for French service
  }
    ... ...
}
```

Figure 2.17. The aspect implementation of multilingualization in AspectJ

```
interface Callback {
  void drawOnePixel(int x, int y, Color c);
}
aspect PixelDataGetting {
  after(int x, int y, Color c) returning (c):
   withincode(BookStore.calcuration(..))
   && args(x, y)
   && call(Color BookStore.calcurateOnePixel(..)) {
    if (callback == null) {
      Registry registry =
        getRegistry("client", 1099);
      callback = (Callback)
        registry.lookup("callback");
    }
    callback.drawOnePixel(xpos, ypos, c);
  }
    ... ...
}
```

Figure 2.18. The class and aspect programs of visualization for a raytracing application in AspectJ

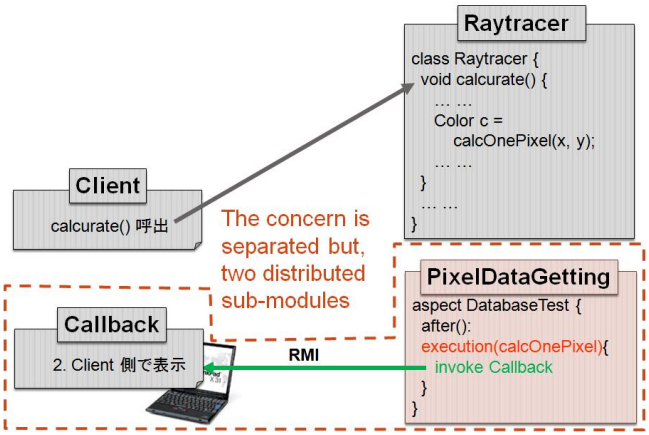

Figure 2.19. A function in AspectJ

Table 2.2. Summary of existing programming languages

existing technologies	(A)	(B)	(C)	(D)
OOP languages (Java)	×	×	×	○
OOP langugaes + delegation	△	×	×	○
Inheritance	△	×	×	○
OOP languages + dynamic scoping	△	○	○	×
AOP languages (AspectJ)	△	○	△	○

Chapter 3

GluonJ

This chapter presents a new AOP language, which is more suitable than existing programming languages for agile software development. It is named *GluonJ* [24, 13]. This language is based on AspectJ, which is one of general-purpose AOP langauges for Java. It provides language constructs for pointcut-advice and an inter-type declaration as well as AspectJ [1].

The reason why GluonJ is based on AspectJ is because AspectJ is the most suitable language for agile software development in existing programming languages. AspectJ satisfies four ideal features shown in Section 2.2 to a certain degree. See the table 2.2 of Section 2.8 again. AspectJ allows software engineers to easily delete the feature that was implemented as an aspect without editing the existing program (feature (B)). Also it is an extension to Java, which is a statically typed language, for AOP (feature (D)). However, AspectJ is not good enough for agile software development. It does not satisfy the features (A) and (C) of an ideal language shown in Section 2.2 too much.

The main contribution of this thesis is that we propose two new language constructs for AOP. These language constructs are named *dynamic refinement* and *remote pointcut*. GluonJ provides these language

[1] In this thesis an inter-type declaration is called refinement.

constructs. To solve the limitations of AspectJ, GluonJ was developed. GluonJ satisfies not only features (B) and (D) but also (A) and (C).

It has been known that AOP is useful for agile software development [39]. However, AOP languages have not been used on agile software development yet. This is because there would not be existing AOP languages that are suitable for agile software development. In this thesis we analized the features of agile software development. We showed that an AOP language is the most suitable languages for agile software development in existing programming languages with two example scenarios. However, existing AOP languages such as AspectJ are not good enough for agile software development since they do not satisfy the features (A) and (C). Thus we proposed dynamic refinement and remote pointcut.

GluonJ is also an extended Java language. Our extension to Java is small. GluonJ uses annotations and thereby does not extend the lexical syntax of Java. It exploits the type system of Java as much as possible. Thanks to these, a GluonJ program can be developed on a normal Java IDE (Integrated Development Environment). Particularly, software engineers can enjoy the coding supports by the normal IDE even for GluonJ programming. A GluonJ program is compiled by a normal Java compiler. Only a special runtime system is needed to run a GluonJ program. We introduced these features for industrial acceptability, where software engineers tend to stay with existing tools.

3.1 New Language Constructs for AOP

Unlike AspectJ, GluonJ provides two new language constructs for AOP. These constructs are *dynamic refinement* [53] and *remote pointcut* [54]. GluonJ that provides dynamic refinement and remote pointcut satisfies the features (C) and (A) in Section 2.2 more than existing programming languages. In the rest of this section, we explain motivation and concept of these language constructs. The detail of them will be explain in Chapter 4 and 5.

3.1.1 Dynamic Refinement

We proposes dynamic refinement, which allows software engineers to dynamically refine the definition of an existing class. By using dynamic refinement in GluonJ, software engineers can redefine existing methods

and append new methods, fields, and interfaces to an existing class according to dynamic contexts. Since these changes are described in a separate component (or module), this language mechanism is useful for separation of concerns.

GluonJ that provides dynamic refinement satisfies the feature (C) of an ideal language for agile software development in Section 2.2. Dynamic refinement allows software engineers to separate newly appended functions from the rest of the program and other functions as aspects. A software engineer can develop his function without consideration of different functions that other engineers develop at same iteration. Also it is easy to merge developed functions into an existing program. Unlike cflow pointcut in AspectJ, dynamic refinement in GluonJ provides a mechanism for grouping multiple redefined methods according to a same dynamic context.

The concept of dynamic refinement is similar to the mechanism of dynamic scoping. Generally speaking, a language that has dynamic scoping is not easy-to-use. Such a language decreases the readability and understandability of its program. However, when software engineers, in parallel, improve a software product on agile software development, dynamic scoping is often effective. Thus GluonJ exploits dynamic scoping.

To make dynamic refinement available, we added only small extension to the original Java. Dynamic refinement is described with annotations within the standard Java syntax. Hence, a GluonJ program can be developed on existing Java IDEs such as Eclipse [5] and NetBeans [9]. It can be edited with a normal Java editor and compiled with a normal Java compiler. Only a special runtime system is needed to run a GluonJ program. A bytecode compiled by a normal Java compiler is transformed by a custom class loader of GluonJ.

Using the standard Java and normal Java IDEs is worthy of challenge with respect to industrial acceptability. Java is a statically typed language and hence enabling dynamic refinement is more difficult than in dynamically typed languages such as Python and Ruby. Nevertheless, Java is one of the most widely used languages in industry and providing an IDE-support is mandatory today. Although it is also a possible option to provide a custom GluonJ compiler and an IDE for GluonJ, for example, an Eclipse plugin for GluonJ, developing those compiler and IDE with industrial-strength quality needs a large amount of effort. If we can enable dynamic refinement within the standard Java syntax on normal Java IDEs, this approach is more desirable.

Moreover, appended methods and fields are recognized by a normal Java IDE. For example, the Eclipse IDE provides coding support called *code assist*. When a programmer types a variable name and then a period (.), Eclipse pops up a list of available methods and fields on that variable so that she can easily complete typing an expression. In GluonJ, the list popped up by Eclipse includes the methods and fields appended by dynamic refinement. This is because we have carefully designed GluonJ so that a class extended by dynamic refinment will be represented by a normal Java type and appended methods and fields will belong to that type.

Although GluonJ allows software engineers to dynamically appending a method to an existing class, a GluonJ program never throws a NoSuchMethodException if it is successfully compiled and loaded. A naive implementation of dynamic refinement would wrongly allow a client to call an unavailable method, which will be appended later by refinement but not now. To avoid such a wrong call, which will throw a runtime exception, GluonJ requires programmers to follow some programming conventions. A GluonJ program satisfying these conventions never calls an unavailable method. If a program does not satisfy the conventions, it is statically detected before the program starts running. To do this, GluonJ exploits the type system of Java and the class loader of GluonJ.

GluonJ allows software engineers to dynamically refine an existing class during runtime to a certain degree. It allows applying and removing refinement to/from a class on demand. Naively designed dynamic refinement may allow a call to a method that has not been appended yet or that has been already removed and then it may cause a runtime type error. However, a GluonJ program never causes such a runtime type error as a NoSuchMethodException although it may fail an explicit type cast and throw a ClassCastException. To guarantee this property, GluonJ does not use a custom type checker. It exploits the type checking by a normal Java compiler and the verification by the custom class loader of GluonJ.

3.1.2 Remote Pointcut

We propose remote pointcut, which is new AOP language constructs for distributed computing. A remote pointcut is a function for designating join points in the execution of a program running on a remote host. Although a pointcut in AspectJ identifies execution points on the local

host, a remote pointcut can identify them on a remote host. In other words, when the thread of control reaches the join points identified by a remote pointcut, the advice body associated with that remote pointcut is executed on a remote host different from the one where those join points occur. Remote pointcuts are analogous to remote method calls, which invoke the execution of a method body on a remote host.

GluonJ that provides remote pointcut satisfies the feature (A) of an ideal language for agile software development in Section 2.2. This language construct can simplify the code of a component implementing a feature in distributed software as an aspect. AspectJ is a useful programming language is a useful programming language for developing distributed software. However, even if a feature can be implemented as a single aspect at the code level, it might need to be deployed on different hosts and it would therefore consist of several sub-components or subprocesses running on each host. Remote pointcut enables implementing such a feature as simple, non-distirbuted component without concerns about network processing.

The aspect weaving in GluonJ is performed at load time on each participating host. The normal Java classes on each participating host must be loaded by the class loader provided by GluonJ. This class loader weaves aspects and classes on the fly. The compiled aspects are stored in the aspect server. The parts of the compiled code except for the advice bodies are automatically distributed by the aspect server to each host, so the latest aspects can be woven when the classes are loaded. The software engineers of GluonJ do not have to manually deploy the compiled aspects to every host.

We also propose another language construct named remote refinement, which allows software engineers to declare a new method and field in an existing class on a remote host. An aspect can declare that it will respond to certain methods and field-access requests on behalf of other objects. In GluonJ, these methods and fields can be declared other objects on multiple remote hosts. Since the description of the refinement automatically distributed from the aspect server to every host, declaring a method or field to classes on remote hosts is simple. The software engineers only have to install the compiled aspect on the aspect server. Unlike in AspectJ, they do not have to manually deploy the woven aspect and classes to every host.

Moreover GluonJ provides a mechanism for hot deployment. This function is for automatically reloading the classfiles of an application

running on top of the runtime system by Remote GluonJ. The developers of Remote GluonJ do not have to shutdown a working program since they apply new aspects to the program on its runtime system. Before the users start up their application on top of the runtime system provided by Remote GluonJ, they must put the classfiles on a directory specified by the runtime system. When the classfiles of aspects and classes on the specific directory are changed, all runtime systems automatically restart distributed software.

3.2 An Overview of GluonJ

GluonJ is a Java annotation based language and implemented the extension to Java without changing the original Java syntax. This lets developers write a GluonJ program with the standard Java IDE like Eclipse IDE. This section explains an overview of GluonJ.

Java's annotations are convenient language constructs for implementing a language extension without changing the lexical syntax of Java. For example, AspectJ5 [11] allows programmers to describe aspects in regular Java. An extended language constructs of AspectJ5, such as a pointcut, is described by using annotations.

However, language extensions implemented with annotations are not understood by a normal Java IDE (Integrated Development Environment). Programmers must use an extended IDE for those language extensions, for example, the AspectJ plugin for Eclipse IDE if they want to enjoy IDE support for the language extensions.

Our extended version of Java also uses Java annotations for implementing a refinement mechanism. Unlike other naive implementations, however, our implementation is quite compatible to a normal Java IDE. We have carefully designed Dynamic GluonJ so that programmers can exploit coding support by a normal Java IDE. For example, a new method appended by a refinement is listed in the available methods shown by the code assist of Eclipse IDE.

3.2.1 Refinement

Software evolution is one of the most significant topics in software industry. To react to altering and evolving requirements at a rapid pace,

software must be extended quickly. To minimize this effort, the extensions should be implemented in a modular fashion as much as possible.

Refinement is a language mechanism for extending an existing class. It is an useful mechanism for component-based programming and thus a number of languages with refinement or similar mechanisms, such as expander and intertype declaration, have been proposed [79, 19, 70, 43, 16, 2]. Some languages allow only static refinement but others allow even dynamic refinement. The latters such as CaesarJ [16] enable programmers to partly modify the definition of an existing class during runtime. This is useful for programming an object that changes its behavior according to dynamic contexts. Refinement allows programmers to implement such behavior as a separate component for each context and compose desired software from a selected set of components.

Note that refinement is different from subclassing or naive mixin. Refinement directly modifies the definition of an *original* class that the refinement targets for extension. On the other hand, subclassing produces an extended definition of a super class while preserving the original definition of the super class. Therefore, to use a method overwritten in a subclass, a client program must explicitly creates an instance of the subclass. The appended method is available only on an instance of the subclass. It is not on an instance of the super class. If a class is extended by refinement, the original definition of that class is completely replaced with the new extended definition. To use a method overwritten by refinement, a client program does not have to be modified. All the subclasses of the class extended by refinement inherit the extended definition without any program modification.

The GluonJ programming language provides a refinement mechanism. Current AOP languages such as AspectJ are not perfectly suitable to implement the extension to existing classes. In the case of AspectJ, intertype declarations for appending getter/setter methods to existing classes are natural presentation whereas around advices for overriding existing methods by the execution pointcut are not. The expressive power of the around advices is somewhat too powerful. Although what we want to do is to define a modified version of the method, the advice body is not executed as a method on the target object; it is executed on an instance of the aspect.

If a simple thing should be expressible in a simple form, then we can use a refinement mechanism such as Mixin Layers [70]. This mechanism allows developers to describe an extension as if it were a subclass ex-

tending the target class. Unlike a subclass, however, it directly modifies the definition of the target class. All calls to the original methods are replaced with calls to the modified versions of those methods.

Although refinement enables intuitive presentation, its expressive power is obviously lower than the pointcut-advice mechanism. In some cases, using pointcut-advice instead of refinement achieves better modularization and presentation [15]. However, introducing both pointcut-advice and refinement into a single AOP language is not acceptable because it degrades the conceptual integrity of the programming model [65]. It will increase the amount of the language specification, make the learning curve steeper, and complicate programming activities. It would be also difficult to keep the semantic consistency between the two mechanisms from the viewpoint of language design.

Changing an existing method Refinements that are provided by GluonJ allow writing an extension to the definition of an original class. They are written in Java as subclasses of their original classes. Unlike the inheritance and mixin mechanisms, however, refinements directly modify the original class definition. Thus to use the extended class definition, its clients do not need to explicitly make a new instance of the extended class. Such subclasses representing refinements are called *refinement classes*. They are language constructs available within aspect declarations of GluonJ. Aspects are also written in Java. We here explain the specification of refinement classes on details.

To change an existing method in an original class, programmers overwrite it in a refinement class described as a subclass of the original class. For example, to change the implementation of the getPrice method in the Book class:

```
public class Book { // original class
  protected String title;
  protected int price;
  public Book(String t, int p) {
    title = t;
    price = p;
  }
  public String getTitle() {
    return title;
  }
```

```
  public int getPrice() {
    return price;
  }
}
```

programmers must write the following refinement class BookLogger:

```
@Glue class Logging {
  @Refine public static class BookLogger
                       extends Book {
    @Override public int getPrice() {
      System.out.println(price);
      return super.getPrice();
    }
  }
}
```

The BookLogger class is a normal Java class that extends the original class Book and overrides the getPrice method. Note that the refinement class is a static nested class in the Logging class annotated with @Glue. Such a class is called *an @Glue class* and can group refinement classes related to each other.

A method in a refinement class can use super for invoking the methods directly declared in its original class. The semantics of super is the same as for subclassing.

Once the two classes are compiled and loaded, the runtime system of GluonJ automatically modifies the original class Book. Thus, during runtime, a Book object is an instance of the modified version of Book, the behavior of which is equivalent to that of an instance of BookLogger in the regular Java. For example,

```
Book b = new Book("AspectJ Primer", 50);
int p = b.getPrice();
```

The call to getPrice invokes the method declared in the BookLogger class because the original method in Book has been replaced. Thus, the mehtod prints the value of price before returning it. Note that refinement is different from subclassing. Although a subclass can partly modify the definition of its super class, an instance of the subclass must be explicitly created for using the modified class definition. Moreover,

a subclass does not affect the behavior of the sibling classes that share the super class with that subclass. On the other hand, a refinement class affects the behavior of its sibling classes. It directly modifies the original class (*i.e.* the super class) and hence the sibling classes inherit the modified definition from their super class.

From the viewpoint of a Java IDE, the definition of a refinement class is just a normal subclass. Hence, programmers can fully exploit coding support provided by the IDE. For example, they can add @Override to a method in a refinement class. If the name or the signature of that method is wrong and the method does not override a method in the original class, the IDE will report an error. Recall that @Override in the regular Java declares that the method with this annotation overrides a method in its super class.

Although a refinement class looks like a normal subclass, creating an instance of a refinement class is prohibited in GluonJ. Creating an array of instances of a refinement class is also prohibited. On the other hand, using a refinement class as a type name is allowed. We will later discuss details of this issue.

Appending new methods To append a new method to an original class, programmers declare the new method in a refinement class extending that original class. For example, the following refinement class BookPrinter appends a new method print to the Book class:

```
@Glue class Printing {
  @Refine public static class BookPrinter
                       extneds Book {
    public void print() {
      System.out.println(
        "Book[" + title + "," + price + "]");
    }
  }
}
```

The original Book class is automatically modified by the runtime system of GluonJ at load time according to Printering. The behavior of instances of the modified Book class in GluonJ is equivalent to that of instances of BookPrinter in the regular Java.

To call a method appended by refinement, a reference to the target object must be cast to the type of the refinement class. For example, the

following code calls the print method appended by the refinement class BookPrinter above:

```
public void printBook(Book b) {
  ((BookPrinter)b).print();
}
```

The type cast from Book to BookPrinter always succeeds while the refinement class BookPrinter is applied to the Book class. Programmers can understand this programming convention by the analogy to down-cast in Java. To call a method in a subclass, a reference to the target object must be down-cast to the subclass type. This type cast succeeds only if the target object is an instance of that subclass. Although a refinement class is not equivalent to a subclass, this analogy would help programmers understand the semantics of GluonJ.

The method print appended by the refinement class is included in the list of the available methods shown by Eclipse (see Figure 3.1) when a programmer types the period at the end of the following sequence:

```
((BookPrinter)b).
```

This is because Eclipse recognizes BookPrinter as a normal class, which declares the print method under the interpretation for the regular Java. Although GluonJ extends the semantics of Java by using annotations, the lexical representaion of a GluonJ program can be read as a normal Java program and the types recognized by this reading are almost equivalent to the types under the semantics of GluonJ. We have carefully designed GluonJ to preserve this property so that the coding support by Java IDEs will be useful for writing a GluonJ program.

In GluonJ, the type cast from the type of an array of an original class to the type of an array of its refinement class succeeds while the refinement class is applied to the original class. For example, the following test cast succeeds.

```
Book[] books = new Book[8];
  :
BookPrinter[] printers = (BookPrinter[])books;
```

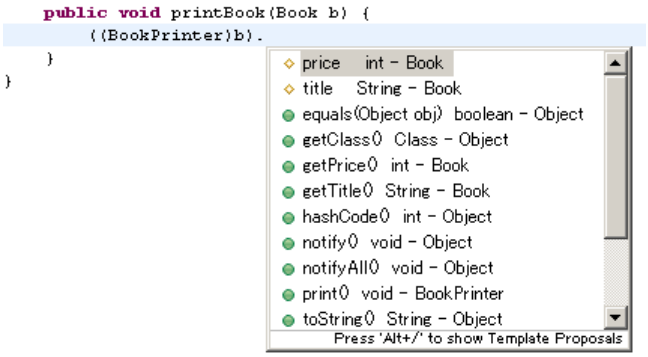

Figure 3.1. The code assist of Eclipse pops up a list of available methods and fields on the b variable. Not only existing methods in Book but also print() newly appended by BookPrinter are included in that list.

Overriding methods with private A refinement class allows users to override private methods in its original class only if it is annotated by @Privileged. For example, suppose that a private method validateBookStore has been declared in BookStore.

```
@Glue class TraceAspect {
  @Refine @Privileged
  static abstract class BookStoreTracer extends BookStore {
    abstract void super_validateBookStore();
    private void validateBookStore() {
      System.out.println("validate");
      super_validateBookStore();
    }
  }
}
```

In the example above, the method validateBookStore in BookStoreTracer is substituted for the original method validateBookStore in BookStore. To call the private method validateBookStore in the original BookStore from the refinement class, an abstract method named super_validateBookStore must be declared in the refinement class. The method name must start with super_, which is followed by the name of the private method in the original class. @Privileged gives an aspect the power enough to violate the information hiding principle. Dynamic GluonJ adopts @Privileged because it is necessary to write white-box tests, for example.

Changing an existing static method Unlike a subclass, a refinement class can override a static method in its original class. For example, the following class changes the implementation of the static createBook in its original class Book:

```
@Glue class Logging {
  @Refine public static class BookLogger
                       extends Book {
    public static Book createBook(String title,
                                  int price) {
      System.out.println("create a book");
      return Book.createBook(title, price);
    }
  }
}
```

A call to the createBook method in Book invokes the implementation above in BookLogger. For example,

```
Book.createBook("AspectJ Primer", 50);
```

this method call first prints the message "create a book" and then executes the original implementation declared in Book. An overriding method in a refinement class can invoke the overridden static method in its original class as in the example above.

Appending new interfaces In GluonJ, to append a new interface to an original class, the interface is declared in a refinement class. For example, the following refinement class makes the Book class implement the Tracer interface:

```
@Glue class Tracing {
  @Refine public static class BookTracer
                       extneds Book
                    implements Tracer {
    public void trace(String msg) {
      System.out.println("[TRACE] " + msg);
    }
  }
}
```

```
public interface Tracer {
  void trace(String msg);
}
```

This refinement class also appends the trace method, which is declared in the Tracer interface. To call the trace method, programmers must cast a reference to the target object to the BookTracer class or the Tracer interface.

Appending new fields GluonJ also enables appending a new field to an original class. In GluonJ, to append a new field to an original class, the field is declared in a refinement class. For example, the following refinement class directly appends a field counter and a method count to the Book class:

```
@Glue class Counting {
  @Refine public static class BookCounter
                      extends Book
                      implements Counter {
    int counter = 0;
    public int getPrice() {
      counter++;
      return super.getPrice();
    }
    public int count() {
      return counter;
    }
  }
}
public interface Counter {
  int count();
}
```

This refinement class BookCounter counts how many the method getPrice calls on the Book object. When getPrice is called, the value of the newly appended counter increases. To check the value of the counter, the users call a method count in an interface Counter.

The order of aspect weaving If there are several refinement classes for extending the same target class, each of them is woven with the target class in turn in the order of the priority that the users specify. The priority is specified by using @Include.

For example, suppose that two aspects that extend BookStore are declared as following:

```
@Glue class LowerPriorityAspect {
  @Include HigherPriorityAspect higherAspect;
  @Refine static class RefinedBookStore1 extends BookStore {
    protected Book findTitle(String title) {
      return super.findTitle(title);
    }
  }
}
@Glue class HigherPriorityAspect {
  @Refine static class RefinedBookStore2 extends BookStore {
    protected Book findTitle(String title) {
      return super.findTitle(title);
    }
  }
}
```

The field higherAspect with @Include is declared in the aspect LowerPriorityAspect. @Include specifies another aspect included. HigherPriorityAspect is included in LowerPriorityAspect as a child aspect. When they are woven, a child aspect with a higher priority is woven first. The parent aspect, which contains a field with @Include, has a lower-priority. Hence HigherPriorityAspect has a higher priority over LowerPriorityAspect.

A refinement class in an aspect that has a higher priority extends an original class before a refinement class in a lower-priority aspect. BookStore is first modified by RefinedBookStore2. The modified BookStore is then modified by RefinedBookStore1. When the original method findTitle in BookStore is called, the method findTitle in RefinedBookStore1 is invoked. super.findTitle() within RefinedBookStore1 then invokes findTitle in RefinedBookStore2. Finally, a call super.findTitle() within RefinedBookStore2 invokes the original one in BookStore.

3.2.2 Pointcut-advice

Another kind of member of an @Glue class is a pointcut-advice, which is a field annotated by @Before, @After, or @Around. Unlike AspectJ, a pointcut and an advice are not separated. A @Glue class can contain any number of pointcut-advices as an @Refine class.

A field representing a pointcut-advice, which we below call *a pointcut field*, must have the type Pointcut. An @Glue class can contain any number of fields of the type Pointcut and it can annotate only some of them by @Before *etc.* If fields of the Pointcut type are not annotated, then they are not treated as pointcut fields.

A typical pointcut field is like this:

```
@Before("{ System.out.println(‘call!‘); }")
Pointcut pc = Pcd.call("figure.Point#setX(..)");
```

Note that a back-quote ‘ used in an argument to @Before is interpreted as an escaped double quote ". This notation is provided for convenience. The declaration above specifies that the following block statement:

```
{ System.out.println("call!"); }
```

is executed when a setX method in figure.Point is called. Since the annotation is @Before, the statement is executed just before the method body starts running, after all the arguments to the method are evaluated. The block statement above is below called *an advice body*.

A Pointcut object specifies when a block statement passed to @Before is executed. The Pcd (Pointcut Designator) class provides factory methods to create a Pointcut object. The call method returns a Pointcut object specifies the time when a method is called. The String argument to call:

```
figure.Point#setX(..)
```

specifies a call to say declared in figure.Point. # is a separator between a class name and a method name. The parameter types of the say method are not specified since (..) is given. The argument above specifies calls to setX(), setX(int), setX(int,int),...

Pointcut designators GluonJ can deal with several kinds of pointcuts. The following is a list of factory methods in Pcd and Pointcut:

- Pointcut call(String *methodPattern*)

  When a method (or a constructor) specified by methodPattern is called.

- Pointcut get(String *fieldPattern*)

  When the value of a field specified by fieldPattern is obtained.

- Pointcut set(String *fieldPattern*)

  When a new value is assigned to a field specified by fieldPattern.

- Pointcut within(String *classPattern*)

  When a thread of control is within a method immediately declared in the class specified by classPattern.

- Pointcut within(String *methodPattern*)

  When a thread of control is within the body of a method specified by methodPattern.

- Pointcut annotate(String *annotationPattern*)

  When a method or a field with an annotation specified by annotationPattern is accessed.

- Pointcut when(String *javaExpression*)

  While javaExpression is true. This corresponds to AspectJ's if pointcut designator.

- Pointcut cflow(String *methodPattern*)

  While a method specified by methodPattern is running.

The pointcuts except call, get, and set are usually used with one of these three poitncuts call, get, or set. These pointcuts can be composed with others by using .and or .or. For example,

```
Pointcut pc =
  Pcd.call("figure.Point#setX(..)").and.within("test.PointTest");
```

The created Pointcut object specifies the time when the setX method in figure.Point is called from a method declared in the test.PointTest class.

Multiple .and and .or can be used in a single expression. For example,

```
Pointcut pc =
  Pcd.call("figure.Point#setX(..)").and
     .within("test.PointTest").or
     .call("figure.Point#getX());
```

This specifies the time when the setX method is called within the test.PointTest class or when the getX method of the figure.Point class is called within any class. .and has a higher precedence than .or. If you want to change this precedence rule, you must use the expr method in Pcd and Pointcut. This method works as parentheses.

```
Pointcut pc =
  Pcd.expr(Pcd.call("figure.Point#setX(..)").or
              .call("figure.Point#getX()")).and
     .within("test.PointTest")
```

This specifies the time when either setX or getX of an figure.Point object is called within test.PointTest. The two call pointcuts are grouped by the expr method.

The expr method is available in the middle of an expression:

```
Pointcut pc =
  Pcd.within("test.PointTest").and.
     expr(Pcd.call("figure.Point#setX(..)").or
             .call("figure.Point#getX()"))
```

You can also split the declaration above into two:

```
Pointcut pc0 =
  Pcd.call("figure.Point#setX(..)").or
     .call("figure.Point#getX());

@Before("{ System.out.println(‘call!‘); }")
Pointcut pc =
  Pcd.expr(pc0).and.within("test.PointTest");
```

This also creates the same Pointcut object. Since the declaration of pc0 is not annotated, pc0 is not treated as a pointcut field. It is only used as a sort of temporary variable.

For negation, GluonJ provides .not, which can be placed after Pcd, .and, or .or. .not has the highest precedence. For example,

```
Pointcut pc =
  Pcd.call("figure.Point#setX(..)").and
     .not.within("test.PointTest");
Pointcut pc2 =
  Pcd.not.within("test.PointTest").and
     .call("figure.Point#setX(..)");
```

Both the two pointcuts above specify the time when the setX method is called within any class except test.PointTest.

call takes a method pattern. It is a concatenation of a class name, a method name, and a list of parameter types. The class name and the method name are separated by #. For example,

```
figure.Point#move(int, int)
```

represents a move method declared in figure.Point. It receives two int parameters. The class name must be a fully qualified name, which includes a package name. If the class name ends with +, subclasses of that class also match the pattern. A class name and a method name can include a wild card *. For example, figure.* means any class in the figure package. A list of parameter types cannot include a wild card *. To specify any type, (..) is used.

A method pattern may match a constructor if the method name in the pattern is new. For example,

```
figure.Point#new(int, int)
```

This pattern matches a new expression with two int arguments.

A field pattern taken by get and set is similar to a method pattern. It is a concatenation of a class name and a field name. For example,

```
figure.Point#x
```

represents a x field declared in figure.Point. A class name and a field name can include a wild card * as well.

Finally, a class pattern is a fully-qualified class name. It can include a wild card *. An annotation pattern is a fully-qualified class name starting with/without @, for example, @figure.Change. It can include a wild card *.

Advice body The declaration of a pointcut field is annotated by either @Before, @After, or @Around. If @Before is used, the advice body passed to that annotation as an argument is executed just before the time specified by the pointcut field. If @After is used, the advice body is executed just after the time specified by the pointcut field. For example, if the pointcut field specifies the time when a method is called, then the advice body is executed just after a return statement in the body of the called method is executed.

- @Before(*adviceBody*)

  The given advice body is executed before the time specified by the pointcut field annotated by this.

- @After(*adviceBody*)

  The given advice body is executed after the time specified by the pointcut field annotated by this.

- @Around(*adviceBody*)

  The given advice body is executed instead of the computation specified by the pointcut field annotated by this.

@Around has a special meaning. If it is used, the given advice body is executed instead of the code fragment specified by the pointcut field annotated by that @Around. For example,

```
@Around("{ System.out.println(‘call!‘); }")
Pointcut pc =
  Pcd.call("figure.Point#setX(..)").and
     .within("test.PointTest");
```

If a method in test.PointTest class attempts to invoke the setX method, that method invocation is intercepted. Then the body of the setX method is not executed but instead the advice body given to the @Around is executed. In other words, the advice body is substituted for the call to say from a method in test.PointTest.

Runtime Contexts In an advice body given to @Around, a special form $proceed is available. It represents the original computation that the advice body is substituted for.

```
@Around("{
  System.out.println(‘call!‘); $_ = $proceed($$);
}")
Pointcut pc =
  Pcd.call("figure.Point#setX(..)");
```

This advice body first prints a message and then invokes the setX method in figure.Point since the method invocation is the original computation that the advice body is substituted for. $proceed is used as a method name. If it is called, it executes the original computation, that is, the say method. $$ represents the original set of arguments given to the computation, that is, the setX method.

$_ is the special variable that the resulting value must be stored. Since @Around substitutes the computation specified by a pointcut field, its advice body must set $_ to an appropriate value before finishing. The type of $_ is the same type as the resulting value of the original computation. If the return type of the originally called method is void, then the value stored in $_ is ignored. Otherwise, the value stored in $_ is used as the result of the advice body. For example, suppose that the getX method returns a value of the int type:

```
int x = p.getX();
```

If the pointcut field:

```
@Around("{ $_ = 100; }")
Pointcut pc =
  Pcd.call("figure.Point#getX()");
```

is applied to the call to getX, then the value stored in $_, which is 100, is assigned to the variable x. This is because the advice body is substituted for the call to getX. The getX method is never executed.

The meanings of $proceed, $$, and $_, depend on the kind of the original computation. However, the following statement:

```
$_ = $proceed($$);
```

executes the original computation whatever is the original computation.

If the original computation is a method call, then $proceed executes that method call. It takes the same set of parameters as the original method and returns the same type of value. $$ represents the list of the original arguments. The type of $_ is the return type of the original method. The value stored in $_ is used as the result of the advice body.

If the original computation is a field read, $proceed executes that operation. It takes no parameter and returns the value of the field. $$ represents an empty list. The type of $_ is the same as the field type. The value stored in $_ is used as the result of the field access instead of the value of the accessed field.

If the original computation is a field write, $proceed changes the value of the field. It takes a new value as a parameter and returns no value (*i.e.* void). $$ represents the value that the original computation attempts to assign. $_ is still available but the value stored in $_ is ignored.

Although $$ represents a list of actual arguments given to original computation, the value of an individual argument is also accessible through a special variable. If original computation is a method call, then $1 represents the first argument, $2 represents the second argument, and so on. $0 represents the target object that the method is called on. Thus, $0.getClass() returns the type of the target object. The name of the method is available from $name. To obtain a reference to the caller/accessor object, *i.e.* an issuer of original computation, this should be used.

A value assigned to $1, $2, ... is reflected on $$. For example,

```
@Around("{ $1 = 50; $_ = $proceed($$); }")
Pointcut pc =
  Pcd.call("figure.Point#setX(int)");
```

After this advice body is executed, the value stored in $_ is the value returned from the setX method called with 50. Thus, the effects of the advice body above is equivalent to:

```
@Around("{ $_ = $proceed(50); }")
Pointcut pc =
  Pcd.call("figure.Point#setX(int)");
```

$0, $1, $2, ... are available within an advice body given to @Before or @After.

Chapter 4

Dynamic Refinement

This chapter presents the detail of dynamic refinement. Dynamic refinement allows software engineers to dynamically refine a class definition to a certain degree. It allows them to dynamically inactivate a refinement that has been applied to a class C and activate another refinement for the same class C. This function improves the description of aspects.

This chapter also presents how dynamic refinement is implemented for Java. Since Java is a statically typed language, it is not obvious how to implement a mechanism for switching refinements during runtime. Our idea of the implementation is to generate a class definition that reflects a union of all refinements.

The current version of GluonJ does not have dynamic refinement. To append the function of dynamic refinement to GluonJ later, we remodelled the implementation of GluonJ and prototyped dynamic refinement. To discriminate between the remodelled GluonJ and regular GluonJ, the remodelled one is called *Dynamic GluonJ*. In the rest of this chapter, we explain the detail of dynamic refinement and Dynamic GluonJ that provides a language construct for it.

4.1 Design Overview of Dynamic GluonJ

Programmers of Dynamic GluonJ can use a normal Java IDE (Integrated Development Environment) to exploit coding support by the IDE. This is significant for the industrial acceptability of a new language. A program of Dynamic GluonJ is written in the regular Java with Java's annotations. A new class definition refined in Dynamic GluonJ was carefully designed so that the IDE can recognize that refinement and reflect it on the coding support such as the code assist of Eclipse. Moreover, a Dynamic GluonJ program never throws a runtime exception reporting that an undefined method is called. Guaranteeing this fact is not straightforward because Dynamic GluonJ allows programmers to refine a class definition during runtime.

In Dynamic GluonJ, a refinement, which represents modifications to an original class, is described as a subclass of the original class. The subclass is annotated with @Refine and called *a refinement class*. Both the original class and the refinement class are compiled by a normal Java compiler. Then the runtime system of Dynamic GluonJ actually modifies the definition of the original class at load time according to the specification given by the refinement class.

4.1.1 Changing an existing method

To change an existing method in an original class, programmers overwrite it in a refinement class described as a subclass of the original class. For example, to change the implementation of the getPrice method in the Book class:

```
public class Book { // original class
  protected String title;
  protected int price;
  public Book(String t, int p) {
    title = t;
    price = p;
  }
  public String getTitle() {
    return title;
  }
  public int getPrice() {
```

```
    return price;
  }
}
```

programmers must write the following refinement class BookLogger:

```
@Glue class Logging {
  @Refine public static class BookLogger
                       extends Book {
    @Override public int getPrice() {
      System.out.println(price);
      return super.getPrice();
    }
  }
}
```

The BookLogger class is a normal Java class that extends the original class Book and overrides the getPrice method. Note that the refinement class is a static nested class in the Logging class annotated with @Glue. Such a class is called *an @Glue class* and can group refinement classes related to each other.

A method in a refinement class can use super for invoking the methods directly declared in its original class. The semantics of super is the same as for subclassing.

Once the two classes are compiled and loaded, the runtime system of Dynamic GluonJ automatically modifies the original class Book. Thus, during runtime, a Book object is an instance of the modified version of Book, the behavior of which is equivalent to that of an instance of BookLogger in the regular Java. For example,

```
Book b = new Book("AspectJ Primer", 50);
int p = b.getPrice();
```

The call to getPrice invokes the method declared in the BookLogger class because the original method in Book has been replaced. Thus, the mehtod prints the value of price before returning it. Note that refinement is different from subclassing. Although a subclass can partly modify the definition of its super class, an instance of the subclass must be explicitly created for using the modified class definition. Moreover, a subclass does not affect the behavior of the sibling classes that share

the super class with that subclass. On the other hand, a refinement class affects the behavior of its sibling classes. It directly modifies the original class (*i.e.* the super class) and hence the sibling classes inherit the modified definition from their super class.

From the viewpoint of a Java IDE, the definition of a refinement class is just a normal subclass. Hence, programmers can fully exploit coding support provided by the IDE. For example, they can add @Override to a method in a refinement class. If the name or the signature of that method is wrong and the method does not override a method in the original class, the IDE will report an error. Recall that @Override in the regular Java declares that the method with this annotation overrides a method in its super class.

Although a refinement class looks like a normal subclass, creating an instance of a refinement class is prohibited in Dynamic GluonJ. Creating an array of instances of a refinement class is also prohibited. On the other hand, using a refinement class as a type name is allowed. We will later discuss details of this issue.

4.1.2 Appending new methods

To append a new method to an original class, programmers declare the new method in a refinement class extending that original class. For example, the following refinement class BookPrinter appends a new method print to the Book class:

```
@Glue class Printing {
  @Refine public static class BookPrinter
                       extneds Book {
    public void print() {
      System.out.println(
        "Book[" + title + "," + price + "]");
    }
  }
}
```

The original Book class is automatically modified by the runtime system of Dynamic GluonJ at load time according to Printering. The behavior of instances of the modified Book class in Dynamic GluonJ is equivalent to that of instances of BookPrinter in the regular Java.

To call a method appended by refinement, a reference to the target object must be cast to the type of the refinement class. For example, the following code calls the print method appended by the refinement class BookPrinter above:

```
public void printBook(Book b) {
  ((BookPrinter)b).print();
}
```

The type cast from Book to BookPrinter always succeeds while the refinement class BookPrinter is applied to the Book class. Programmers can understand this programming convention by the analogy to down-cast in Java. To call a method in a subclass, a reference to the target object must be down-cast to the subclass type. This type cast succeeds only if the target object is an instance of that subclass. Although a refinement class is not equivalent to a subclass, this analogy would help programmers understand the semantics of Dynamic GluonJ.

The method print appended by the refinement class is included in the list of the available methods shown by Eclipse (see Figure 4.1) when a programmer types the period at the end of the following sequence:

```
((BookPrinter)b).
```

This is because Eclipse recognizes BookPrinter as a normal class, which declares the print method under the interpretation for the regular Java. Although Dynamic GluonJ extends the semantics of Java by using annotations, the lexical representaion of a Dynamic GluonJ program can be read as a normal Java program and the types recognized by this reading are almost equivalent to the types under the semantics of Dynamic GluonJ. We have carefully designed Dynamic GluonJ to preserve this property so that the coding support by Java IDEs will be useful for writing a Dynamic GluonJ program.

In Dynamic GluonJ, the type cast from the type of an array of an original class to the type of an array of its refinement class succeeds while the refinement class is applied to the original class. For example, the following test cast succeeds.

```
Book[] books = new Book[8];
  :
BookPrinter[] printers = (BookPrinter[])books;
```

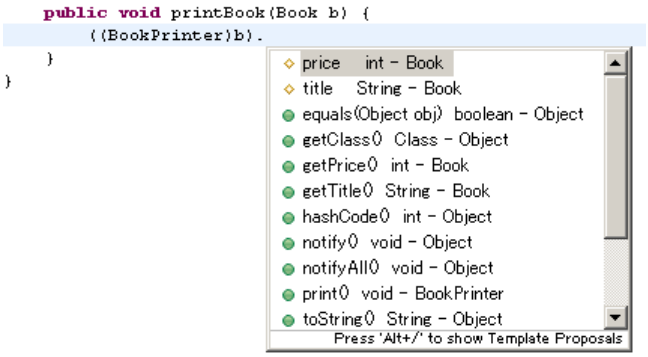

Figure 4.1. The newly appended method is included in the list of avaiable methods shown in a code assist by Eclipse

4.1.3 Appending new interfaces

In Dynamic GluonJ, to append a new interface to an original class, the interface is declared in a refinement class. For example, the following refinement class makes the Book class implement the Tracer interface:

```
@Glue class Tracing {
  @Refine public static class BookTracer
                      extneds Book
                    implements Tracer {
    public void trace(String msg) {
      System.out.println("[TRACE] " + msg);
    }
  }
}
public interface Tracer {
  void trace(String msg);
}
```

This refinement class also appends the trace method, which is declared in the Tracer interface. To call the trace method, programmers must cast a reference to the target object to the BookTracer class or the Tracer interface.

4.1.4 Changing an existing static method

Unlike a subclass, a refinement class can override a static method in its original class. For example, the following class changes the implementation of the static createBook in its original class Book:

```
@Glue class Logging {
  @Refine public static class BookLogger
                       extends Book {
    public static Book createBook(String title,
                                  int price) {
      System.out.println("create a book");
      return Book.createBook(title, price);
    }
  }
}
```

A call to the createBook method in Book invokes the implementation above in BookLogger. For example,

```
Book.createBook("GPCE 2007", 50);
```

this method call first prints the message "create a book" and then executes the original implementation declared in Book. An overriding method in a refinement class can invoke the overridden static method in its original class as in the example above.

4.1.5 Dynamic refinement

Dynamic GluonJ allows applying a refinement class to its original class during only limited time. Recall that a refinement class is a static nested class in a class annotated with @Glue, which can group multiple refinement classes related to each other. If the @Glue class is annotated with @Cflow, then all the refinement class included in that @Glue class are activated during only the time specified by the @Cflow annotation. The extension by the refinement classes is effective during only that time.

The argument to @Cflow is the time while refinements are effective. For example, the following refinement class appends the print method only while the getBookPrice method in the BookStore class is executed.

```
@Cflow("void BookStore.getBookPrice(Book)")
@Glue public class CflowBasedPrinting {
  @Refine public static class BookPrinter
                      extends Book {
    public void print() {
      System.out.println(
          "Book[" + title + "," + price + "]");
    }
  }
}
```

The signature of a method is given as the argument to @Cflow[1]. An @Glue class with @Cflow is active only while the specified method is executed. After that, the @Glue class is inactivated. A refinement class included in the @Glue class is applied to its original class only during the active time of the @Glue class.

A method appended by a refinement class with @Cflow is never invoked when the refinement class is inactive. Recall that, to call a method appended by a refinement class, a reference to the target object must be cast to the type of the refinement class. The type cast from the type of an original class to the type of the refinement class with @Cflow succeeds only when the refinement class is active. If an overriding method in a refinement class with @Cflow is invoked when the refinement class is inactive, the overridden method in its original class is invoked instead of the overriding one.

The following code calls the print method appended by the refinement class BookPrinter shown above:

```
public class BookStore {
  public int getBookPrice(Book b) {
      ((BookPrinter)b).print();
      return b.getPrice();
  }
  public int getBookTitle(Book b) {
      // throw ClassCastException
      ((BookPrinter)b).print();
      return b.getTitle();
```

[1]There is restriction on the signature of a method that can be specified as the argument to @Cflow. The restriction will be shown in Section ??.

```
  }
}
```

The refinement class BookPrinter is effective only during the execution of the getBookPrice method. Thus, it is not effective while the getBookTitle method is executed. Thus, the cast from Book to BookPrinter fails and then causes a ClassCastException. The print method on b is never invoked.

The extension to a class by a refinement class with @Cflow is applied to all the instances of that class during the specified time. It is applied to not only existing instances but also instances created while the @Glue class is active. Note that this application is per-thread basis. When other threads access the instances, the extension by the refinement class is not effective. The effect of the @Glue class is within the thread that activated the @Glue class.

An @Glue class without an @Cflow annotation is always active during runtime. A refinement class included in that @Glue class is statically applied to its original class.

4.1.6 Restrictions by @Cflow

To prevent a method appended by dynamic refinement from being called when the method is not available, Dynamic GluonJ requires programmers to follow the programming conventions shown below.

Let G is an @Glue class associated with @Cflow and let G include a refinement class R. R appends an interface I to its original class. If a type T is either R, I, or an array type of R or I, such as $R[]$ and $I[]$, then the following rules must be satisfied:

1. A field of the type T appears only within G.
2. The type T is not the return type or one of the parameter types of the method specified as the argument to @Cflow.
3. T is not an exception type (*i.e.* a subclass of Throwable). It is not the parameter to a catch clause.
4. The refinement class R does not override a static method in its original class.

We later discuss why these rules guarantee that a method appended by dynamic refinement is never invoked when the method is not available. For more details, see Section 4.2.5.

The runtime system of Dynamic GluonJ checks whether or not these rules are satisfied in a given Dynamic GluonJ program when the program is loaded. If the rules are not satisfied, the runtime system throws an exception. Otherwise, the program is successfully loaded and it will never throw a NoSuchMethodException, which reports that an unavailable method is called. NoSuchMethodException is an exception type included in the standard Java library. It is thrown when a client calls a method that does not exist.

Note that a Dynamic GluonJ program may throw a ClassCastException when an object reference is cast from an origianl class to its refinement class. However, once the cast operation succeeds, a Dynamic GluonJ program never throws a NoSuchMethodException.

4.1.7 The order of @Glue class application

If there are several refinement classes for modifying the same original class, each refinement class is applied to the original class by the runtime system of Dynamic GluonJ in turn in the order of the priority that the programmers specify. The priority is specified by using @Include.

For example, suppose that two @Glue classes that modify BookStore are declared as following:

```
@Glue class LowerPriorityGlue {
  @Include HigherPriorityGlue higherGlue;
  @Refine public static class RefinedBookStore1
                       extends BookStore {
    public int getBookPrice(Book b) {
      return super.getBookPrice(b);
    }
  }
}
@Glue class HigherPriorityGlue {
  @Refine public static class RefinedBookStore2
                       extends BookStore {
    public int getBookPrice(Book b) {
      return super.getBookPrice(b);
```

```
    }
  }
}
```

The field higherGlue with @Include is declared in the @Glue class LowerPriorityGlue. @Include specifies another @Glue class included. The @Glue class HigherPriorityGlue is included in LowerPriorityGlue as a child @Glue class. When they are applied to its original class, a child @Glue class with a higher priority is applied first. The parent @Glue class, which contains a field with @Include, has a lower-priority. Hence HigherPriorityGlue has a higher priority over LowerPriorityGlue.

A refinement class in the @Glue class that has a higher priority is applied to its original class before a refinement class in a lower-priority @Glue class. BookStore is first modified by RefinedBookStore2. Then the modified BookStore is modified by RefinedBookStore1. When the original method getPrice in BookStore is called, the method getBookPrice in RefinedBookStore1 is invoked. super.getBookPrice() within RefinedBookStore1 then invokes getBookPrice in RefinedBookStore2. Finally, a call super.getBookPrice() within RefinedBookStore2 invokes the original one in BookStore.

If there are several refinement classes with @Cflow for modifying the same class, each refinement class is applied to its original class in turn in the order of the priority specified by @Include. The modified behavior of objects of that class is changed according to whether or not those refinement classes with @Cflow are active.

4.1.8 Changing the initial values of existing fields

Refinement allows changing the initial value of an existing field declared in its original class like subclassing. For example, the following refinement class sets the initial value of the protected field enableDebug in the Book class to true:

```
@Glue class Debugging {
  @Refine public static class BookDebugger
                       extends Book {
    protected boolean enableDebug = true;
  }
}
```

The enableDebug field in BookDebugger overrides the initial value of the original enableDebug field with the same name and signature in the Book class. enableDebug is set to true by refinement class when a constructor in the Book class is called.

4.1.9 Appending new fields

To append a new field to an original class, the field can be described in its refinement class like subclassing. Note that the field is visible only within methods declared in the refinement class. For example, the following refinement class appends a new field ps of the type of PrintStream to the Book class:

```
@Glue class Logging {
  @Refine public static class BookLogger
                      extends Book {
    public PrintStream ps =
      new PrintStream(new File("log.txt"));
    @Override public int getPrice() {
      ps.println("executes getPrice()");
      return super.getPrice();
    }
  }
}
```

The refinement class BookLogger also overrides the getPrice method in Book. Thus, A call to getPrice in Book first writes the message "executes getPrice()" to the file named log.txt and then executes the original one in the Book class.

4.1.10 Changing and calling private methods

A normal refinement class cannot change a private mehtod in its original class to preserve the principle of the information hiding. However, a refinement class annotated with @Privileged can do it. Suppose that a private method validate has been declared in Book:

```
@Glue class Logging {
  @Refine @Privileged
```

```
  public static abstract class BookLogger
                       extends Book {
    abstract void super_validate();
    private void validate() {
      System.out.println("executes validate");
      super_validate();
    }
  }
}
```

This refinement class changes the validate method. If this method is called on a Book object, the validate method declared in BookLogger is executed. To call the original validate method in Book from BookLogger, an abstract method super_validate must be declared in the refinement class BookLogger. If the super_validate method is called, the original validate method in Book is executed. The name of the abstract method must start with super_, which is followed by the name of the private method that programmers want to call.

Unlike private methods, the current version of Dynamic GluonJ does not allow changing a final method declared in an original class. Even a @Privileged refinement class cannot do so. Enabling this is part of our future work.

4.1.11 Accessing private fields

A private field can be accessed only in a refinement class with @Privileged. For example, suppose that a private field stock has been declared in the Book class:

```
@Glue class StockControl {
  @Refine @Privileged
  public static class BookStockClerk extends Book {
    private int stock;
    public boolean inStock() {
      return stock != 0;
    }
  }
}
```

The refinement class BookStockClerk appends a new method inStock to Book. To access a private field stock within inStock, a private field with the same name must be declared in the refinement class. If a refinement class declares a private field with the same name as a private field in its original class, that field is identical to the private field in the original class.

4.2 Implementation Issues

To parse the arguments of annotations such as @Refine and @Cflow during runtime and then extend a program of Dynamic GluonJ according to the annotations, the program runs on the runtime system provided by Dynamic GluonJ. The runtime system transforms the bytecode of a program of Dynamic GluonJ at load time. A program of Dynamic GluonJ must be compiled by a standard Java compiler in advance. Our implementation of refinements generates an original class definition that is the union of all refinements for that class at the bytecode level. The runtime system uses Javassist [23, 25], which is a toolkit for transforming Java bytecode. This section presents the processing by the runtime system of Dynamic GluonJ in detail.

4.2.1 Appending new methods

When Dynamic GluonJ finds a method that a refinement class appends to an original class, it copies the appended method to the original class at load time. A call to a method in the original class on super within the copied method is replaced with the method call on this.

When there are several refinement classes that append a new method with the same name to an original class, each appending method is copied to the original class in turn in the order of their priority. Dynamic GluonJ first copies an appending method with the highest priority to its original class. Next it renames the appended method with the highest priority to a unique name and then copies a newly appending method with a lower priority to the original class. The priority of refinement classes was mentioned in Section 4.1.11.

4.2.2 Changing methods in an original class

If a refinement class includes a method changing a method existing in its original class, Dynamic GluonJ first renames the overridden (changed) method in the original class to a unique name. Then it copies the overriding (changing) method in the refinement class to the original class. A call to the original method on super within the overriding method is replaced with a call on this to the renamed method. A call to a non-overridden method in the original class on super within the overriding method is also replaced with the method call on this.

When there are several refinement classes change the same method in its original class, each overriding method is copied to the original class according to their priority. First Dynamic GluonJ renames the overridden method to a unique name and then copies an overriding method with the highest priority to the original class. Next it changes the name of the copied method with the highest priority to a unique name and then copies an overriding method with a lower priority to its original class. A call to the original method on super within the copied method with a lower priority is replaced with the call to the method copied with a higher priority. The priority of refinement classes was mentioned in Section 4.1.11.

4.2.3 @Glue classes with @Cflow

When Dynamic GluonJ copies an overriding method in a refinement class with @Cflow to its original class, it inserts a check code at the beginning of the overriding method. This code checks whether or not the refinement class with @Cflow is active, and if it is not, it forwards the call to the overridden method in its original class. Since methods in multiple refinement classes with @Cflow may override a method in its original class, if the refinement class with @Cflow is not active, the check code forwards to the next method overridden by another refinement class according to their priorities. When there are several refinement classes with @Cflow that append a method with the same name to an original class, the check code is also inserted in each appending method like changing original methods.

4.2.4 Type names, type cast, and instanceof

The runtime system of Dynamic GluonJ searches compiled class files (Java bytecode) for replacing all occurrences of the type name of refinement classes with their original class names. The replaced type names are ones used as a return type, a parameter type, or a thrown exception type. The runtime system substitutes the bytecode generated by the runtime system for virtual machine instructions that perform a runtime type check. These instructions are checkcast (type cast operator) and instanceof.

If the destination type of the checkcast instruction is a refinement class with @Cflow, the bytecode generated by the runtime system is inserted before the instruction. The bytecode checks whether or not the refinement class with @Cflow is active. If the refinement class is inactive, ClassCastException is thrown. Also, the same bytecode is inserted if the destination type of the checkcast instruction is an array type of a refinement class with @Cflow.

If the right-hand type of the instanceof instruction is a refinement class with @Cflow, the instanceof instruction is replaced with the bytecode generated by the runtime system. The bytecode first checks whether or not the refinement class with @Cflow is active. If not, the bytecode pushes 0 (false) onto the operand stack. Otherwise, it really executes the instanceof instruction and pushes the result onto the stack. Also, the instanceof instruction is replaced if the right-hand type is an array type of a refinement class.

The virtual machine instructions new, anewarray (array creation), and aastore (assignment to an array element) are not replaced by the runtime system. In Dynamic GluonJ, creating an instance of a refinement class and an array of it is prohibited. Thus, since the new and anewarray instructions for a refinement class do not appear in the whole program, they do not need to be replaced by the runtime system. Also, aastore is not replaced although it performs a runtime type check. See the following example:

```
R[] r = new R[1];
C[] c = r;
c[0] = new C();     // throws ArrayStoreException
```

Here, R is a subclass of C. The last assignment, which is compiled into the aastore instruction, performs a runtime type check and throws an

exception because of the covariant rule of Java. If R is a refinement class and C is its original class, then the behavior of the aastore instruction would be extended to perform a type check considering dynamic activation. However, since creating an array of a refinement class is prohibited in Dynamic GluonJ, there is never an array that contains only R objects. Thus, replacing the aastore instruction is unnecessary.

4.2.5 Why is no NoSuchMethodException thrown?

A Dynamic GluonJ program successfully compiled and loaded never throws a NoSuchMethodException. This exception is included in the standard Java library and it is thrown when an undefined method is called. Suppose that a client class calls a method m declared in a class C. After the client class is compiled, if the method m is removed from the class C and the class C is separately compiled again, then the client class will throw a NoSuchMethodException during runtime.

It is not straightforward to guarantee that a Dynamic GluonJ program does not throw a NoSuchMethodException. If @Cflow is specified, a refinement class is effective during only limited time. Thus, a method appended by the refinement class is not always available. If a client class calls that method when it is not available, then an exception will be thrown.

Our idea is the following. (1) To call a method appended by a refinement class, a reference to the target object first must be cast to the type T of that refinement class. Then, (2) a value of this type T is alive at most while the refinement class is effective. Therefore, a method appended by a refinement class is never called when it is not available.

The property (1) is enforced by a normal Java compiler and the cast operation extended by the runtime system of Dynamic GluonJ. Since Java is a statically typed language, it is not possible that a client calls a method appended by a refinement class if the target reference is not the type of the refinement class. The type cast to the refinement class is extended by Dynamic GluonJ and it succeeds only when the refinement class is effective.

(2) is guaranteed by the programming conventions shown in Section 4.1.6. A value obtained by the type cast to a refinement class may be stored in a local variable or a field. However, if the field is not declared in the refinement class, the value stored in the field will survive after the refinement class becomes ineffective. A method call on that

field may throw a NoSuchMethodException. To avoid this, a Dynamic GluonJ program must follow the programming conventions, which guarantee that a field of the type of a refinement class is declared only in its @Glue class. When the @Glue class becomes inactive, thus, that field also becomes invisible from the program. The programming conventions also guarantee that a value of the type of a refinement class does not leak out of its @Glue class as a return value or an exception thrown out.

4.3 Example Programs

4.3.1 Multilingualization for an Online Book Store

Here we illustrate the implementation of multilingualization shown in Section 2.3 in Dynamic GluonJ. The program in Figure 4.3.1 is an overview of Japanese service for the online book store. The code in Figure 4.3.1 is French service. See the Figure 4.3.1. The program in Dynamic GluonJ can be separated from the rest of the program such as BookStore, French service. Thus the developer of Japanese service can be written without consideration of French service.

Unlike AspectJ, Dynamic GluonJ provides a mechanism for grouping multiple redefined methods according to a same dynamic context. In the implementation in Dynamic GluonJ, there is no redundant repetition of @Cflow. A refinement class can be grouped into an aspect with other refinement classes that share the same @Cflow. This grouping mechanism makes the description of an aspect simple and intuitive if the implementation of a crosscutting concern is a homogeneous aspect, in which each pointcut is associated with a different advice. A number of advices can be grouped according to the structure of the target program woven with the aspect, so that the readability of the aspect will be improved. Furthermore, since dynamic refinement in Dynamic GluonJ uses the same syntax as regular classes, each advice (*i.e.* a method in a refinement class) looks like a regular method. This will also improve the readability.

The aspect FrenchService in Figure 4.3.1 can also be separated from the rest of the program and other feature (Japanese service).

```
@Glue class JapaneseService {
  @Refine static class MultilingualizedBookStore
              extends BookStore {
    String searchBookTitlesForJapanese(String kw) {
      return searchBookTitles(kw);
    }
  }
  @Cflow("String BookStore#searchBookTitleForJapanese(..)")
  @Refine static class JapaneseService
              extends BookStore {
    int getPrice(Book b) {
      return proceed() * 105;
    }
    String getTitle(Book b) {
      // ... ...
    }
    String getCurrency(Book b) {
      // ... ...
    }
  }
}
```

Figure 4.2. The implementation of Japanese service for multilingualization in Dynamic GluonJ

4.3.2 A Test Code for Online Book Store with Mock Objects

A regression test is a good application of Dynamic GluonJ. Regression testing is to confirm whether or not all the functionality that previously worked still correctly works after the tested program is modified. Since performing a regression test as frequently as possible is one of today's best practice, automating the test is strong demand in industry.

Writing a test program is, however, not a simple task. Suppose that we want to check the behavior of the registerCustomer method in the following BookStore class:

```
public class BookStore {
  public BookStore() {}
```

```
@Glue class FrenchService {
  @Refine static class MultilingualizedBookStore
               extends BookStore {
    String searchBookTitlesForFrench(String kw) {
      return searchBookTitles(kw);
    }
  }
  @Cflow("String BookStore#searchBookTitleForFrench(..)")
  @Refine static class FrenchService
               extends BookStore {
    int getPrice(Book b) {
      return proceed() * 7 / 10;
    }
    String getTitle(Book b) {
      // ... ...
    }
    String getCurrency(Book b) {
      // ... ...
    }
  }
}
```

Figure 4.3. The implementation of French service for multilingualization in Dynamic GluonJ

```
  protected HashMap customers;
  public void registerCustomer(String customer,
                               String password) {
    if (customers == null)
      customers = new HashMap();

    customers.put(customer, password);
  }
  public boolean login(String customer,
                       String password) {
    // If customer makes the login succeed, then
    // this method returns true.  Otherwise,
    // it returns false.
  }
```

```
    :
}
```

This method records a pair of a customer's name and her password. To test the behavior of this method, the readers might think that a test program first calls registerCustomer method and then checks whether or not the login method returns true if a right pair of a customer's name and her password is given.

However, in practice, we often want not to call the login method for the test because this method might also perform some setup for database accesses done later or some undesirable side-effects. The method might pop up a GUI window and require a user to press a button on it. In these cases, the simplest way for testing the registerCustomer method is to inspect the customers field of the BookStore object and confirm that the pair is correctly recorded after the method is called.

If there is no accessor method to the customers field, we must append it to the BookStore class for the test but appending an accessor only for testing is not acceptable because this means a change of the specification of BookStore. Other parts of the program might accidentally call the appended accessor method, which was not included in the original specification.

A refinement class of Dynamic GluonJ enables programmers to append such an accessor method and make it available only while a specific test method is running. The tested program does not have to be modified at all. The following refinement class appends an accessor method containsCustomer to BookStore only while a test method is executed:

```
@Cflow("void BookStoreTest.testRegisterCustomer()")
@Glue class RegisterCustomerTesting {
  @Refine public static class CustomersAccessor
                       extends BookStore {
    public
    boolean containsCustomer(String customer) {
      return customers.containsKey(customer);
    }
  }
}
public class BookStoreTest extends TestCase {
  BookStore store = Factory.makeBookStore();
  public void testRegisterCustomer() {
```

```
    assertFalse(((CustomersAccessor)store)
                    .containsCustomer("muga"));
    store.registerCustomer("muga", "xxxx");
    assertTrue(((CustomersAccessor)store)
                    .containsCustomer("muga"));
  }
    :
}
```

The above test class BookStoreTest is used with the JUnit framework [56][2]. store is a field declared in the test class and initialized during the constructor call[3]. The containsCustomer method appended by the refinement class is called before and after the invocation of registerCustomer within the testRegisterCustomer method.

The above refinement class appends an accessor method for accessing the protected customers field to the BookStore class. Even if the field is private, a refinement class of Dynamic GluonJ can append it. How an appended method accesses a private field was mentioned in Section 4.1.11.

4.4 Effectiveness

This section discuss when and where dynamic refinement is effectively used.

Dynamic refinement is effectively used for extensions to existing classes according to session objects in web applications. A software engineer of a web application often appends new interfaces (methods) for creating session objects to the business logic of the web application. Once a user of the web application accesses an interface via a network, a session object that responds to the accessed interface is created. A software engineer uses the created session object and implements the program for switching processings that a user wants according to its object. Since

[2] TestCase is a class provided by JUnit. assertFalse and assertTrue are methods inherited from TestCase. assertFalse throws an exception unless the first argument is false. assertTrue throws an exception if the first argument is not true.

[3] Although the constructor of BookStore in the BookStore class is simple, a real constructor of BookStore might take a complex data structure for the initialization and preparing that data structure might be so tedious that we might want to reuse an existing factory method for the initialization.

those switched processings have same signatures, they can simply be separated as dynamic refinements provided by GluonJ. GluonJ allows several software engineers to develop different processings without writing the codes for switching their processings according to dynamic contexts.

On the other hand, dynamic refinement is ineffectively used for an extension to an existing class according to a fine-grained dynamic context. It enables separating an extesion to an existing class according to any control-flow, no matter how small. However, its implementation decreases the modularity or readability of the whole of the program. This is because a software engineer must often declare multiple extensions to an existing class for multiple small contexts. For example, suppose that a software engineer is extending the behavior of an existing class C according to two dynamic context *d1*, *d2*. By using dynamic refinement, he would write two refinement classes C11, C12 according to *d1*. Also he would describe three refinement classes C21, C22, C23 according to *d2*. Dynamic refinement in GluonJ allows separating these extensions by dynamic contexts as @Cflow refinement classes. However, there are six kinds of the behaviors of the woven class according to *d1* and *d2*.

Also dynamic refinement is effectively used for unit tests. It allows software engineers to switch mock objects that are applied to its target program according to each test method. By using dynamic refinement, also software engineers can easily append accessor methods to a target program only while a specific test method is executed for accessing the internal state of the target object. They do not have to edit the program of the target class. When testing a business logic of a web application, due to an instance creation overhead, the instance of the business logic is not created each test method. The instance is shared with several test methods. In this case, dynamic refinement is useful.

Although refinement in GluonJ allows a software engineer to improve and refactor an existing program at a certain degree, it is difficult to change (override) the signature of an existing method in its class. A language construct for refinement in GluonJ is based on Java subclassing. By using the syntax of subclassing, GluonJ exploits the type system of Java. A GluonJ program can be compiled by the normal Java compiler and be developed on a normal Java IDE. However, Java subclassing cannot override the signature of an existing signature. It is the limitation of Java subclassing and GluonJ.

Classbox/J [19] is an extension to Java for the concept of the classbox [14] and allows switching refinement classes (classboxes) applied to its

existing class according to the type of a caller-side object. By using Classbox/J, a software developer does not need to write the codes for type-checking (*i.e.* cast and instanceof expressions) in a program such as a Java GUI library. However, The current version of GluonJ does not provides a language construct for switching refinements by caller-side objects. It allows switching refinements according to control-flow only.

4.5 Related Work

Most aspect-oriented programming (AOP) languages, frameworks, and systems for Java more or less have a mechanism for change the definition of an original class during runtime. We below mention differences between them and Dynamic GluonJ.

AspectJ [38, 2] is a general-purpose AOP language for Java and it is the extension of Java. Also AspectJ5 [11], AspectWerkz [3], and JBoss AOP [6], which are annotation-based AOP frameworks, allow programmers to write aspects in the regular Java syntax. They provide language constructs for dynamically changing the behavior of an existing method on an object with pointcut-advices. Unlike Dynamic GluonJ, however, they do not provide language constructs for dynamically adding and removing new fields and methods to/from an object. Moreover, since the language constructs of JBoss AOP and AspectWerkz cannot access private fields and methods in its original class, a regression test shown in Section 4.3 cannot be written in them.

CaesarJ [50, 16] is an AOP language providing powerful language constructs for changing a class according to dynamic contexts during runtime[4]. It is the extension of Java. To change the behavior of an object during runtime, CaesarJ also allows dynamically activating an aspect that includes pointcut-advices. To dynamically append new methods to an object, it provides a wrapper mechanism. A difference between CaesarJ and Dynamic GluonJ is that Dynamic GluonJ allows writing an extension to its original class in the regular Java syntax since Dynamic GluonJ is an annotation-based framework. Thus a program in Dynamic GluonJ can be developed on an existing IDE. This is good for industrial acceptability. On the other hand, a program of CaesarJ must be written according to programming conventions of CaesarJ. If programmers want to extend an existing Java application with an aspect of CaesarJ,

[4]CaesarJ is also known as a feature-oriented programming language for Java.

they must rewrite the existing application according to the conventions. Moreover, to add new methods to an object of its class, CaesarJ creates a wrapper object with the new methods and binds the original object to it. The wrapper object is not identical to its original object. Refinement of Dynamic GluonJ does not create such a wrapper object for adding new methods to an object. They directly add the methods to the object.

AspectJ and CaesarJ provide Eclipse plugins for customizing Eclipse IDE. The plugins enable those programmers to develop a program in those languages on the customized Eclipse. The programmers can exploit coding support such as a code assist provided by the customized Eclipse. However, the Eclipse plugins cannot extend other IDEs because the specification of plugin machanisms that other IDEs provide is different from Eclipse's. To use coding support of various IDEs when programmers write a program of AspectJ or CaesarJ, AspectJ and CaesarJ must provide the plugins for customizing those IDEs. A program of Dynamic GluonJ can be developed on any normal Java IDEs.

Several frameworks and systems for dynamic weaving for Java have been proposed and developed [64, 63, 75, 69, 21]. Dynamic weaving is a mechanism to dynamically enable weaving/unweaving aspects and allows programmers to change an original class according to aspects during runtime. Most of them have been implemented on top of JPDA [72] or Java Hotswap [30]. Since JPDA captures relevant execution points and intercepts the program execution, pointcut-advice frameworks and systems can easily be implemented on top of it. However, it is difficult to design and implement a mechanism for adding and removing new methods, fields, and interfaces to an original class dynamically. On the other hand, Java Hotswap allows dynamically reloading a class definition. However, since the definition of a newly loaded class must preserve the type signature of its original class, the loaded class definition can only change the implementation of methods declared in its original class. Thus, Java Hotswap cannot also add and remove methods and fields to the original class like JPDA. Another implementation of dynamic weaving is to extend the Java virtual machine (JVM). Steamloom [21] is a powerful dynamic AOP system using a customized JVM, which is based on IBM's Jikes RVM [7]. It supports a mechanism for dynamically weaving pointcut-advices only. Thus it cannot dynamically append and remove new methods and fields to an object.

Refinement is similar to mixin layers [70, 22], virtual classes [47], nested inheritance [55], and so on. To extend an original class and an

original class hierarchy, these mechanisms have been proposed before, but they statically extend a program. Here we introduce other languages that provide refinement and compare those languages with our work.

A classbox is one of the concepts for extending classes and it was originally developed with Smalltalk [14]. The Java extension of the concept of the proposed classbox, called Classbox\J [19], was recently proposed. Classbox\J allows not only appending new members such as fields, methods, constructors, and so on, to existing classes but overriding original methods in the classes. The refined classes can be explicitly imported from other classes. Unlike Classbox\J, Dynamic GluonJ does not provide a language construct for appending a new constructor to an original class. Moreover Classbox\J can dynamically control the scope of an extension to an existing class. In other words, it can switch implementations of methods according to caller-side programs that invoke the methods. On the other hand, Dynamic GluonJ dynamically applies and removes several refinement classes to/from its original class according to dynamic contexts, in particular, a control flow. Unlike Classbox\J, a program of Dynamic GluonJ can be developed on an existing IDE without any plugins.

eJava [79] is the extension of Java and provides language constructs for refinement, called expanders. Expanders allow appending new fields, methods, and interfaces to an original class. To call new methods appended by expanders, its programmers import the expanders into caller-side programs. When the programmers append a feature to a class hierarchy that consists of several classes, they can describe the feature that is appended to each original class as an expander. Then the expanders can be grouped as an expander family. The function has powerful expressiveness. However, all expanders statically extend original classes. In other words, eJava does not provide language constructs for dynamically changing an original class. It cannot change the behavior of an object of its extended class during runtime.

Context-based programming is a concept for dynamically extending original classes and enables switching the extension of original classes according to dynamic contexts during runtime. It was originally developed with CLOS [27]. ContextJ [28] is the extension of Java for the concept. It allows programmers to override an existing method in its original class and to change multiple overriding methods according to dynamic contexts. However, it does not allow appending a new method to an original class according to dynamic contexts. Thus it cannot change the behavior

of an object of its original class during runtime. On the other hand, Dynamic GluonJ enables not only overriding a method in its original class but also appending a new method to its original class during runtime.

Hyper/J [59] is a subject-oriented programming (SOP) language for Java. It allows programmers to specify rules for composing multiple features. To reuse and evolve an existing program, Hyper/J can statically divide the existing program into several parts and then merge the divided parts and other programs. Although language constructs provided by Hyper/J are powerful, Hyper/J needs to transform a program statically. The program transformed by Hyper/J cannot be changed during runtime.

Lieberman originally introduced delegation in the framework of a prototype-based object model [45]. An object (child) may have references to other objects (parents). If a method that a child does not have is executed, the extension is automatically forwarded to a method with the same name on its parents. Darwin/Lava [42] and delegation layers [40, 41] support dynamic delegation for Java. Dynamic delegation allows changing the behavior of an object during runtime according to appended parents. This function is similar to a wrapper mechanism of CaesarJ. A child object is not identical to its parent object. Unlike Darwin/Lava and delegation layers, refinement of Dynamic GluonJ does not create another object. They add methods to the object directly.

4.6 Summary

This chapter proposed dynamic refinement and Dynamic GluonJ, in which the programmers can dynamically modify the definition of an existing class according to refinement during runtime. A main contribution of our work is the pragmatic design of a language construct for dynamic refinement. To make refinement available, Dynamic GluonJ uses Java annotations and thereby does not extend the lexical syntax of Java. Unlike other naive implementations, however, our implementation is quite compatible to a normal Java IDE. Dynamic GluonJ has carefully been designed so that the programmers can exploite coding support by a normal Java IDE. Also, a Dynamic GluonJ program can be edited with a normal Java editor and compiled with a normal Java compiler. Only a special runtime system is needed to run a Dynamic GluonJ program. This feature of Dynamic GluonJ is good for industrial acceptability.

Also, this chapter explained that a Dynamic GluonJ program never

occurs a runtime type error NoSuchMethodException if the program is successfully compiled and loaded. To prevent a method appended by dynamic refinement from being called when the method is not available, Dynamic GluonJ requires programmers to several programming conventions. When a Dynamic GluonJ program satisfies the conventions, an unavailable method is never invoked. To guarantee this property, Dynamic GluonJ does not use a custom type checker. It exploits the type checking by a normal Java compiler and the verification by the custom class loader of Dynamic GluonJ.

Chapter 5

Remote Pointcut

This chapter presents new AOP language constructs for distributed computing. These language construct are called *remote pointcut*. Existing AOP languages such as AspectJ are suitable for agile software development however, such languages are not enough for appending a new function to a distributed software product in an iteration of agile development. In AspectJ, several functions that are newly appended to distributed software cannot often be modularized as simple aspects. Rather, such aspects spread over multiple hosts and explicitly communicated across the network. This chapter shows that remote pointcut enables developers to write simple aspects to modularize newly appended functions.

The current version of GluonJ does not have remote pointcut. To append the function of remote pointcut to GluonJ later, we remodelled the implementation of GluonJ and prototyped remote pointcut. To discriminate between the remodelled GluonJ and regular GluonJ, the remodelled one is called *Remote GluonJ*. In the rest of this chapter, we explain the detail of remote pointcut and Remote GluonJ that provides a language construct for it.

5.1 Motivation

In an iteration of agile software development, crosscutting concerns are included in a new function that is added to a distributed software product. Such concerns cannot be modularized as simple aspects in existing AOP languages such as AspectJ. If we use AspectJ to modularize such concerns, the aspects can be some what modular but it often consists of several sub-components distributed on different hosts. They must be manually deployed on each host and the code of these sub-components must include explicit network processing among the sub-components ofr exchanging data sice they cannot have shared variables or fields. These facts compicate the code of the aspect and degrade the benefits of using aspect-oriented programming.

The source of these problems is that the language constructs of AspectJ do not accommodate to distribution or network processing. Combination of AspectJ and an existing framework for distributed software, such as Java RMI (remote method invocation) [73] is not a solution. The existing frameworks extend language constructs for object-orientation, such as method ccalls, so that they can accommodate distribution. They do not support language constructs for aspect-orientation.

AspectJ is a useful programming language for developing distributed software. It enables modular implementation even if some crosscutting concerns are included in the implementation. However, the developers of distributed software must consider the deployment of the executable code. Even if some concerns can be implemented as a single component ("aspect") at the code level, it might need to be deployed on different hosts and it would therefore consist of several sub-components or sub-processes running on each host. Since Java (or AspectJ) does not provide variables or fields that can be shared among multiple hosts, the implementation of such a concern would include complicated network processing for exchanging data among the sub-components.

Programming frameworks such as Java RMI do not solve this problem of complication. Although they make details of network processing implicit and transparent from the programmers' viewpoint, the programmers still must consider distribution and they are forced to implement the concern as a collection of several distributed sub-components exchanging data through remote method calls. The programmers cannot implement such a concern as a simple, non-distributed monolithic component without concerns about network processing. This is never desirable with

respect to aspect orientation since it means that the programmers must be concerned about distribution when implementing a different concern.

5.2 Design Overview of Remote GluonJ

To address the problems of the previous section, we propose *Remote GluonJ*, which is an extension to GluonJ for distributed software. It allows the users to implement a crosscutting concern as an aspect that does not include explicit network processing using Java RMI, even if that concern cuts across multiple components on different hosts.

5.2.1 Remote pointcut

The most significant difference between AspectJ and Remote GluonJ is that Remote GluonJ provides *remote pointcuts*. A remote pointcut is a function for identifying join points in the execution of a program running on a remote host. In other words, when the thread of control reaches the join points identified by a remote pointcut, the advice body associated with that remote pointcut is executed on a remote host different from the one where those join points occur. Remote pointcuts are analogous to remote method calls, which invoke the execution of a method body on a remote host. Unfortunately, GluonJ does not provide such a pointcut. An advice body in GluonJ is executed on the same host as where the join points identified by a pointcut occur.

Remote pointcuts enable implementing a distributed crosscutting concern as a simple, non-distributed component without concerns about network processing. The following is an aspect written in Remote GluonJ, which conforms to the regular GluonJ syntax:

```
@Glue class Logging {
  @Before("{
    System.out.println(‘set x: ‘ + x);
  }")
  Pointcut pc =
    Pcd.define("int", "x", "$1")
       .call("Point#setX(int)");
}
```

This aspect prints a message whenever the setX() method is called on each participating host. The message is printed on a single particular host wherever the setX() method is called.

The pointcut pc in Logging:

```
call("Point#setX(int)")
```

identifies each join point that is a call to the setX() method in the Point class. Unlike pointcuts in GluonJ, however, this pointcut identifies the join points matching the signature on every host even if the advice body is not deployed on the host.

Aspect server

The body of the advice:

```
System.out.println(‘set x: ‘ + x);
```

is executed just before each call to setX(), but it is executed on a host different from the host where the caller thread is running. If the thread of control reaches the join point, it implicitly sends a message through the network to an *aspect server* running on a different host[1] so that the aspect server will execute the advice body. The thread of control that sent the message blocks until the aspect server finishes the execution of the advice body. Since all of the advice bodies are executed by the aspect server on the central host, they can easily exchange values by storing data in the fields defined in the aspect. These fields are locally accessible from the advice bodies. Note that, in AspectJ, the advice body is executed on the same host where the caller thread is running. Thus it may have to explicitly send values through the network to exchange them with other advice bodies executing on other hosts.

Load-time weaving and remote refinement

Remote GluonJ performs load-time weaving. The normal Java classes on each participating host must be loaded by the class loader provided by Remote GluonJ [44]. This class loader weaves aspects and classes on the fly. The compiled aspects are stored in the aspect server. The

[1] Technically, the aspect server might be running on the same host.

Table 5.1. The pointcut designators of Remote GluonJ

designator	join points
within(*TypePattern*)	the join points included in the declaration of the types matching *TypePattern*
target(*Type or Id*)	the join points where the target object is an instance of *Type* or the type of *Id*
args(*Type or Id*, ...)	the join points where the arguments are instances of *Type*s or the types of the *Id*s
call(*Signature*)	the calls to the methods matching *Signature*
execution(*Signature*)	the execution of the methods matching *Signature*
cflow(*Pointcut*)	all join points that occur between the entry and exit of each join point specified by *Pointcut*
host(*Host*)	the join points in the execution on *Host*.

parts of the compiled code except for the advice bodies are automatically distributed by the aspect server to each host, so the latest aspects can be woven when the classes are loaded. The users of Remote GluonJ do not have to manually deploy the compiled aspects to every host.

This fact improves the usefulness of the refinement (formerly called *the introduction*) in Remote GluonJ. An aspect can declare that it will respond to certain methods and field-access requests on behalf of other objects. In Remote GluonJ, these methods and fields can be declared other objects on multiple remote hosts. Since the description of the refinement is automatically distributed from the aspect server to every host, declaring a method or field to classes on remote hosts is simple. The users only have to install the compiled aspect on the aspect server. Unlike in AspectJ, they do not have to manually deploy the woven aspect and classes to every host. This automatic deployment is useful in the context of distributed unit testing. We will revisit this issue in Section 4.

5.2.2 Pointcut designators

The pointcut designators provided by the current implementation of Remote GluonJ are listed in Table 5.1. Most of the pointcut designators are from AspectJ.

A pointcut designator unique to Remote GluonJ is hosts. It identifies

the join points in the execution on the designated hosts. Although Remote GluonJ can deal with all the join points on every participating host, this pointcut designator is used to identify the join points on particular hosts.

For example, the users of Remote GluonJ can describe the following pointcut with the host pointcut designator:

```
Pointcut pc =
  Pcd.call("Point#setX(int)").and
     .expr(Pcd.host("hostId1").or.host("hostId2"));
```

This pointcut identifies join points that are calls to the setX() method in the Point class on the hosts with the names specified by hostId1 or hostId2. HostId1 and hostId2 are parameters given by the users when the program starts running. These runtime parameters allow the developers to avoid embedding particular host names as constants in the source code so that they can enjoy good flexibility.

Remote GluonJ extends the cflow pointcut designator to handle the control flows of distributed software. cflow identifies join points that occur between the start of the method specified by cflow and the return. It identifies only the join points visited by the thread executing the method specified by cflow. In AspectJ, cflow cannot pick out join points on a remote host since the control-flow data needed to implement cflow is stored in a ThreadLocal variable but the ThreadLocal variable is never passed through a network.

Remote GluonJ provides a custom socket class so that the ThreadLocal variable can be passed through a network. If network communication is performed with this custom socket class [74], then cflow can pick out join points on a remote host. For example, if Java RMI is used for network communication, the following program exports a remote object to make it available to receive incoming calls, using the custom socket class:

```
PointImpl p0 = new PointImpl();
Point p
   = (Point) UnicastRemoteObject.exportObject(
                p0, 40000,
                new DJCClientSocketFactory(),
                new DJCServerSocketFactory());
```

This program exports a PointImpl object, which is accessible from a remote host through the Point interface. The DJCClientSocketFactory and

DJCServerSocketFactory classes are the factory classes provided by Remote GluonJ for creating the custom socket. Remote GluonJ also provides a convenient method with which the program shown above can be rewritten as follows:

```
PointImpl p0 = new PointImpl();
Point p
   = (Point) Remote GluonJ.exportObject(p0, 40000);
```

5.2.3 Pointcut parameters

Like AspectJ, Remote GluonJ allows pointcuts to expose the execution context of the join points they identify. For example, the args pointcut designator can expose method parameters and the target pointcut designator can expose the target object. Each part of the exposed context is bound to a pointcut parameter, which is accessible within the body of the advice. For example,

```
Pointcut pc =
  Pcd.define("int", "x", "$1")
     .call("Point.move(int,int)")
```

This setter pointcut exposes the first int-type parameter to the move method through a pointcut parameter x.

In Remote GluonJ, since remote pointcuts identify join points on remote hosts, the pointcut parameters should refer to data on the remote hosts. By default, they refer to a copy of that data constructed on the aspect server. The runtime system of Remote GluonJ first serializes the data on the remote hosts, transfers it through the network, and constructs a copy from the serialized data. The pointcut parameters available in the advice body refer to that copy.

Pointcut parameters can be specified as remote references instead of local references to the copies. If the configuration file specifies that pointcut parameters of class type C are remote references, then the runtime system of Remote GluonJ dynamically generates a proxy class for C. From the implementation viewpoint, the pointcut parameters are made to refer to instances of that proxy class on the aspect server. If the advice body calls a method on that proxy object, then the method is invoked on the master object on the remote host where the join point occurs.

To generate proxy classes, Remote GluonJ uses the replace approach we developed for Addistant [51]. For example, if the remote object associated with a proxy object is a Widget object, then the proxy class is also named Widget. On the aspect server, this proxy class is loaded instead of the original Widget class. The proxy-class generation is performed with our bytecode engineering library Javassist [23, 25].

Remote references are used not only for pointcut parameters but also references to instances of aspects. Normal Java classes can call methods declared in aspects. The references to the instances of the aspects are also remote references implemented using the same approach as for the pointcut parameters. In addition, the parameters of the methods called on the remote object indicated by a remote reference can be also remote references.

5.2.4 @Local advice and @On advice

The implementations of crosscutting concerns in distributed systems do not always involve multiple hosts. Such crosscutting concerns can be implemented without remote pointcuts as a simple non-distributed component. If they are implemented with remote pointcuts, the execution performance is rather worse because of the overheads due to network communication to the aspect server. These crosscutting concerns should be implemented with the aspects provided by GluonJ.

Remote GluonJ therefore provides ones similar to the advice of GluonJ. The developers can specify that copies of an advice are distributed to each participating host and that body of advice is locally executed on the same host as where the join points exist. This type of advice, which is called *an local advice*, are equivalent to the advices available in GluonJ.

For example, to use an local advice, developers declare a pointcut field annotated by @Local as following:

```
@Glue class Logging {
  @Local
  @Before("{ System.out.println(‘local advice‘); }")
  Pointcut pc =
    Pcd.call("Point#setX(int)");
}
```

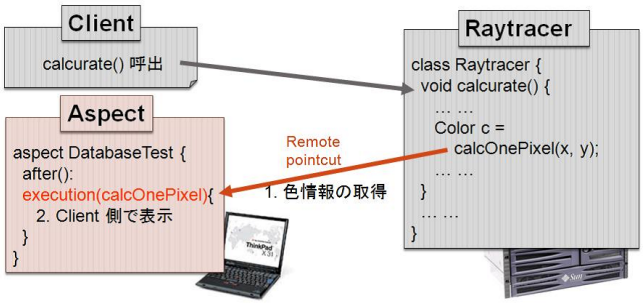

Figure 5.1. The visualization code in Remote GluonJ

The above pointcut allows displaying the log message on the host where specified join point is executed.

Also by using `@On` annotation, an advice can be executed on the specific host.

```
@Glue class Logging {
  @On("hostId")
  @Before("{ System.out.println(‘local advice‘); }")
  Pointcut pc =
    Pcd.call("Point#setX(int)");
}
```

For example, the above pointcut enables executing advice on the host named hostId before the setX method on a Point object is called. hostId is the name of runtime system provided by Remote GluonJ.

5.3 Example Programs

5.3.1 The visualization code for a raytracer program

We illustrate the visualization code that was shown in Section 2.3 in Remote GluonJ. Remote GluonJ allows implementing this function as an simple aspect that runs on client-side (Figure 5.1).

```
@Glue class RealtimeDrawing {
```

```
  @Before("{ RealtimeDrawing.draw(x, y, col); }")
  Pointcut pc =
    Pcd.define("int", "x", "$1")
       .define("int", "y", "$2")
       .define("java.awt.Color", "col", "$3")
       .call("Raytracer#writeOnePixel(..)");

  static void draw(int x, int y, Color col) {
    synchronized (GUIWindow.g) {
      GUIWindow.g.setColor(col);
      GUIWindow.g.drawLine(x, y, x, y);
    }
  }
}
```

The RealtimeDrawing aspect runs on the top of the runtime system provided by Remote GluonJ on a client node. By using remote pointcut, the aspect specifies the execution of a writeOnePixel method on a Raytracer object on different nodes as join points. The advice calls a static method draw in the aspect. draw is a method for drawing the calcurated data on the GUI window of a client program.

5.3.2 A Test Code for Distributed Software

We illustrate an example of unit testing[2] for distributed software. Distributed test code includes crosscutting concerns but, if they are modularized in AspectJ, the code develops the complexities mentioned above. Writing test code for automating unit tests is an important development process that the XP (Extreme Programming) community [37] recommends. The automation results in cleaner code, encourages refactoring, and makes rapid development possible. Recently, simple regression test frameworks such as JUnit and Cactus have been getting popular for the automated unit testing.

[2]Some might think this example should be called not unit testing but end-to-end testing.

5.3.2.1 Unit test for authentication service

As an example, we present test code for a distributed authentication service. The implementation of this service consists of two components: a front-end server AuthServer on a host W and a database server DbServer on another host D. This is a typical architecture for enterprise Web application systems. If a client application needs to register a new user, it remotely calls registerUser() on the front-end server using Java RMI. Then the registerUser() method remotely calls addUser() on the database server, which will actually access the database system to update the user list.

To unit-test the registerUser() method, the test code would first remotely call the registerUser() method and then confirm that the addUser() method is actually executed by the database server. Note that since the test code must confirm that remote method invocation is correctly executed, it must confirm not only that registerUser() on the host W calls addUser() but also that addUser() starts running on the host D after the call.

The test code would be simple and straightforward if the examined program is not distributed. We below show the test code written in AspectJ:

```
aspect AuthServerTest extends TestCase {
  boolean wasAddUserCalled;
  void testRegisterUser() {
    wasAddUserCalled = false;
    String userId = "muga", password = "xxx";
    AuthServer auth = new AuthServer();
    auth.registerUser(userId, password);
    assertTrue(wasAddUserCalled);
  }
  before():
    execution(void DbServer.addUser(String,
                                    String)) {
    wasAddUserCalled = true;
  }
}
```

Although this is not complete code due to the space limitations, the readers would understand the overall structure of the test code. The

main part of the test code is testRegisterUser(). It calls the registerUser() method and then confirms the wasAddUserCalled field is true. This field is set to true by the before advice when the addUser() method is executed.

5.3.2.2 Test code in AspectJ

Unfortunately, the test code becomes more complicated if the examined program is distributed. The test code shown below is a distributed version (again, it is not complete code. Access modifiers such as public and constructors are not shown):

```
// on host T
class AuthServerTest extends TestCase {
  boolean wasAddUserCalled;
  void testRegisterUser() {
    Naming.rebind("test", new RecieverImpl());
    wasAddUserCalled = false;
    String userId = "muga", password = "xxx";
    AuthServer auth
      = (AuthServer) Naming.lookup("auth");
    auth.registerUser(userId, password);
    assertTrue(wasAddUserCalled);
  }
  class ReceiverImpl
      extends UnicastRemoteObject
      implements NotificationReceiver {
    void confirmCall() {
      wasAddUserCalled = true;
    }
  }
}

interface NotificationReceiver
{ // on both hosts
  void confirmCall();
}

aspect Notification { // on host D
  before():
```

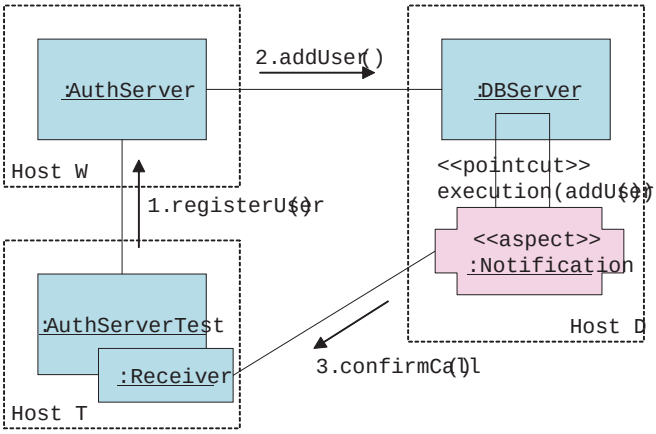

Figure 5.2. The testing code in AspectJ

```
    execution(void DbServer.addUser(String,
                                    String)) {
      NotificationReceiver test
        = (NotificationReceiver)
           Naming.lookup("test");
      test.confirmCall();
  }
}
```

The test code now consists of three sub-components: AuthServerTest, ReceiverImpl, and Notification (Figure 5.2). Although the overall structure is the same, the AuthServerTest and ReceiverImpl objects run on a testing host T but the Notification aspect runs on the host D, where the DbServer is running. The host T is different from W or D.

The testRegisterUser() method on T remotely calls registerUser() on W and then confirms that the wasAddUserCalled field is true. This field is set to true by the confirmCall() method in ReceiverImpl, which is remotely called by the before advice of Notification running on D. The confirmCall() method cannot be defined in AuthServerTest since AuthServerTest must extend the TestCase class whereas Java RMI requires that remotely-accessible classes extends the UnicastRemoteObject class.[3]

[3]This is not precisely accurate. Technically, a confirmCall() can be defined in AuthServerTest by using certain programming tricks. However, the test code would be significantly more complicated.

As we can see, even this simple testing concern is implemented by distributed sub-components and hence we had to write complicated network processing code using Java RMI despite that it is not related to the testing concern. In particular, the Notification aspect is used only for notifying confirmCall() on the host T beyond the network that the thread of control on the host D reaches addUser(). The Notification aspect is a sub-component that are necessary only because confirmCall() and addUser() are deployed on different hosts. This means that the component design of the unit testing is influenced by concerns about distributed.

5.3.2.3 The use of remote pointcut

The testing code was complicated compared to the non-distributed version of the testing code. If we rewrite that testing code in Remote GluonJ, then the resulting code becomes as simple as the non-distributed version:

```
// on host T
@Glue class AuthServerTest extends TestCase {
  static boolean wasAddUserCalled;
  void testRegisterUser() {
    wasAddUserCalled = false;
    String userId = "muga", password = "xxx";
    AuthServer auth
      = (AuthServer) Naming.lookup("auth");
    auth.registerUser(userId, password);
    assertTrue(wasAddUserCalled);
  }
  @Before("{
    AuthServerTest.wasAddUserCalled = true;
  }")
  Pointcut pc =
    Pcd.call("DbServer#addUser(..)");
}
```

Unlike the code in AspectJ, the testing code in Remote GluonJ is not divided into distributed sub-components (Figure 5.3). Although the before advice is executed when the thread of control reaches the addUser() method on the host D, where the DbServer is running, the execution of the before advice is on a different host T, where the testRegisterUser() method is running. Thus the before advice can directly set wasAddUserCalled

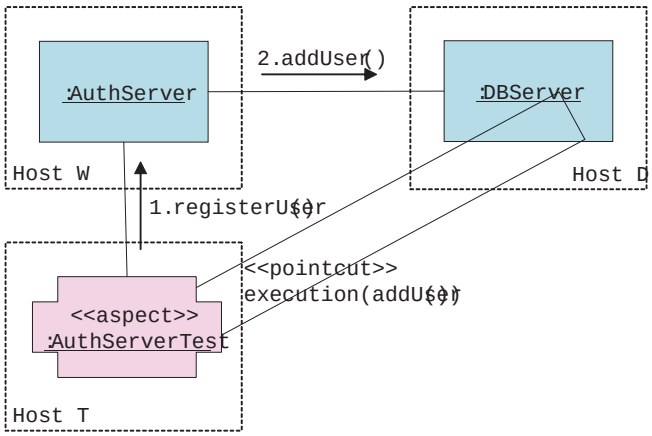

Figure 5.3. The testing code in Remote GluonJ

to true. All the network processing for reporting the execution of the addUser() method to the host T needs not be explicitly described.

Note that the before advice contains the cflow pointcut designator, since Remote GluonJ provides cflow across multiple hosts if the components communicate with the Java RMI. This improves the accuracy of the testing code. The code can examine not only whether or not addUser() is executed, but also whether the caller to addUser() is registerUser().

The testing code in Remote GluonJ has another advantage. Since Remote GluonJ automatically distributes the definitions of the aspects to each participating host and weaves them at load time, the programmers do not have to manually deploy the compiled and woven code to the hosts whenever the definitions of the aspects are changed for different tests.

5.3.2.4 The use of remote refinement

Unit testing sometime requires accessor methods for inspecting the internal state of objects. AspectJ can be used to append such accessor methods just for testing if these methods are not defined in the original program. For example, the developer may want to confirm that the data sent by the registerUser() method is actually stored in the database by the addUser() method. To do this, an accessor method containsUser() must be appended to the DbServer class so that the testing code can examine whether the added user entry is contained in the database.

The remote refinementof Remote GluonJ simplifies such unit test-

ing. If the developers use AspectJ, they have to recompile all the programs and deploy the compiled and woven code to the participating hosts whenever they change the refinement in the aspect. On the other hand, Remote GluonJ can simplify this deployment. Since Remote GluonJ automatically distributes the new definitions of the aspect to the hosts and weaves it at load time, the new aspect is reflected in the programs if the programs are simply restarted.

The following is the testing code written in Remote GluonJ. It appends containsUser() to the DbServer class to. The testRegisterUser() method first confirms that the user muga is not recorded in the database and then it calls the registerUser() method. After that, it confirms that the user muga is recorded in the database.

```
// on host T
@Glue class AuthServerTest extends TestCase {
  void testRegisterUser() {
    String userId = "muga", password = "xxx";
    AuthServer auth
      = (AuthServer) Naming.lookup("auth");
    DbServer db
      = (DbServer) Naming.lookup("db");
    assertTrue(!db.containsUser(userId));
    auth.registerUser(userId, password);
    assertTrue(db.containsUser(userId));
  }
  @Refine static class Diff
              extends DbServer {
    boolean containsUser(String userId) {
    // this method returns true if the user
    // entry specified by userId is found
    // in the database.
  }
}
```

5.4 Experiment

To examine the execution performance of remote pointcuts, we compared the execution time between Remote GluonJ and AspectJ using

Java RMI. For this experiment, we used the testing programs shown in Section 5.3.2.2 (AspectJ using Java RMI) and Section 5.3.2.1 (Remote GluonJ). These programs examine whether registerUser() in AuthServer remotely calls addUser() in DbServer. We measured the elapsed time of the testRegisterUser() method for each program. The body of the addUser() method was empty. In this experiment, the AuthServer and the AuthServerTest ran on the same host while DbServer ran on another host. The AuthServer host was a Sun Blade 1000[4] and the DbServer was a Sun Fire V480[5]. These hosts were connected through a 100 BaseTX network. We used Sun JDK 1.4.0_01 and AspectJ 1.0.6.

Table 5.2. The elapsed time (msec.) of testRegisterUser()

Pointcut parameters	()	(String)	(String,String)
Java + Java RMI	5.9	5.9	6.0
AspectJ + Java RMI	5.9	6.0	6.0
Remote GluonJ	4.8	4.9	5.0
Remote GluonJ without cflow	4.8	4.9	4.9

Table 5.2 lists the results of our measurement. Although the program in Remote GluonJ was slightly faster than in AspectJ, this result does not mean Remote GluonJ is considerably faster than AspectJ using Java RMI. In the program in AspectJ (see Section 5.3.2.2), when the body of the before advice is executed, a remote reference test (lines 31 to 33) is obtained for calling confirmCall(). On the other hand, this remote reference is not obtained in Remote GluonJ during the measurement. It is implicitly obtained by the runtime system in advance. Since obtaining this remote reference needs remote access to the registry server, this difference caused about 1 milli-second ahead of Remote GluonJ in the measurement. We confirmed this fact by other experiment.

The programs shown in Section 5.3.2.2 and 5.3.2.1 do not use pointcut parameters. To evaluate effects by sending pointcut parameters through a network, we also examined the programs in that the before advice (in Remote GluonJ) or the confirmCall() method (in AspectJ) receives one or both of the parameters to the addUser() method (Remote GluonJ). The type of the parameters is the String class. The results

[4]Dual UltraSPARC III 750 MHz with 1 GB of memory and Solaris 8.
[5]UltraSPARC III Cu 900 MHz ×4 with 16 GB of memory and Solaris 9.

of our measurement showed that the performance impacts by pointcut parameters are small.

For fair comparison, we also measured the elapsed time of the program written in Remote GluonJ without cflow since the program in AspectJ did not use cflow. The results were similar to those of the program using cflow since the overhead due to cflow across a network is not significant.

5.5 Implementation Issues

5.5.1 Load-time Weaving

Custom Classloader Remote GluonJ provides unique class loader for implementing remote pointcuts mechanism. As mentioned above (Remote GluonJ runtime on each host doesn't know which join points are identified in pointcuts and which classes are declared in inter-type declarations on startup. Therefore, the class loader provided by Remote GluonJ asks the aspect server those information, receives the information matching join points within a loading class, and weaves (translates) a loading class by using the information.

In the Java language, developers can implement subclass of java.lang.ClassLoader class in order to extend the manner in which the JVM dynamically loads classes [44]. We implement the findClass method defined in ClassLoader class for achieving load-time weaving. The following code is the findClass() method defined in the class loader provided by Remote GluonJ runtime.

```
 1: Class findClass(String className) {
 2:     ...
 3:     byte[] classFile = ...;
 4:     ...
 5:     Pointcut[] pcs = getPointcuts(className);
 6:     InterTypeDecl[] itds = getInterTypeDecls(className);
 7:     ...
 8:     classFile = weaver.weave(classFile, pcs, itds);
 9:     ...
10:     return defineClass( ... );
11: }
```

getPointcuts() is the method that Remote GluonJ runtime ask the aspect server whether join points within the loading class are designated in pointcut or not. It returns an array of Pointcut objects even if join points are designated in pointcut(line 5). GetInterTypeDecl() is also the method that Remote GluonJ runtime ask the aspect server whether the fields and methods of loading class are declared in aspects or not. It returns an array of InterTypeDecl objects even if the fields and methods are declared (line 6). The processing on line 8 is that an *aspect weaver* of Remote GluonJ runtime, the weaver variable, recieves the arguments pcs and itds from the class loader and weaves (translates) the loading class by using the information of pointcuts and inter-type declarations defined in aspects. After weaving the loading class, the aspect weaver returns the array of bytes of loading class to the class loader (line 8). Next, the class loader converts the array of bytes into an instance of class java.lang.Class using the defineClass method defined in class ClassLoader (line 10).

Aspect Weaver The most significant role of weave() in an aspect weaver is to insert the codes for invoking the bodies of advice by bytecode translation at load-time. The aspect weaver is executed by the class loader. Then it modifies the array of bytes of the class by using the Pointcut objects and InterTypeDecl objects passed by the class loader.

The aspect weaver checks whether each of join points within the loading class, such as a method call, a field access, an object creation, and exception event, matches the designated join points in pointcuts or not. Even if a join point within the loading class matches the designated join points, the aspect weaver immediately starts weaving. We show an example of weaving the Line class and Logger aspect as follows:

```
class Line {
  void moveX(int dx) {
    int x1 = p1.getX();
    p1.setX(x1 + dx);
    int x2 = p2.getX();
    p2.setX(x2 + dx);
    observer.paint();
  }
}
@Glue class Logger {
  @Before("{ System.out.println('set x'); }")
```

```
  Pointcut pc =
    Pcd.within("figure.Line").and
       .call("figure.Line#setX(int)")
}
```

In this example, join points designated by Pointcut object is the Point#setX method call. The aspect weaver of Remote GluonJ checks whether each join point within moveX() of the Line class is equal to the Point#setX method call or not. First join point within moveX() is the Point#getX method call. The Point#getX method call is not equal to the designated join point. Next, join point within moveX() is the Point#setX method call. As this join point is equal to the designated join point as pointcut, the weaver inserts the code that invokes the code flagment of @Before before calling Point#setX(). On the same way, the weaver checks whether join points are equal to the designated join point or not. As a result, the weaver modifies the body of moveX() as follow.

```
void moveX(int dx) {
  int x1 = p1.getX();
  Logger.before_$0();
  p1.setX(x1 + dx);
  int x2 = p2.getX();
  Logger.before_$0();
  p2.setX(x2 + dx);
  observer.paint();
}
```

Note that the before_$0 method in Logger is the method that has the code flagment of @Before as a method body. The class loader provided by Remote GluonJ receives the array of modified bytes from the weaver, and then loads those.

5.5.2 Remote References

Unfortunately, any single implementation approach of the proxy-master model cannot deal with all kinds of classes. Each approach convers only the classes satisfying the criteria peculiar to that approach. The developers cannot choose a single approach and enforce the criteria on the whole program. For example, one of the approaches needs to modify the

declaration of the class of master objects. Since the JVM does not accept modified system classes, if an instance of a system class is a remote object, that approach cannot be used. A different approach must be used for that case.

To avoid this problem, Addistant [51], which is our old work, provides several different approaches for implementing a proxy-master model. It provides four approaches; *replace*, *rename*, *subclass*, and *copy*. The developers can choose one from the four for each class of master. The differences among the four approaches are mainly how a proxy class is declared, how caller-side code, that is, expressions of remote method invocations, is modified, and how a master class is modified. The four approaches cover most of case.

Remote GluonJ allows developers to specify a policy of proxy implementaion for each of classes. The developers can declare that every instances of remote reference of the class are implemented the specified approach. The policy declaration is written in a policy file in an XML syntax. Remote GluonJ runtime receives the policy file that the developers write on startup, it parses the file as an XML document. Then, Remote GluonJ runtime automatically implements remote references of each classes on demands at load-time.

As an example, the policy file:

```
<policy>
   <class name="Point" proxy="replace">
   <class name="java.lang.String" proxy="copy">
   <aspect name="Logger" proxy="replace">
</policy>
```

means that every instances of remote reference of the Point class is implemented by using *replace* approach, java.lang.String class is implemented by using *copy* approach, and Logger aspect representing the LoggingAspect aspect is implemented by using *replace* approach. Currently, Remote GluonJ allows to choose one from *replace* and *copy* approaches.

5.5.3 Hot Deployment

Remote GluonJ provides a mechanism for *hot deployment* [6]. This function is for automatically reloading the classfiles of an application running

[6]It has already been provided by most web application servers and web containers such as JBoss[32, 35], Tomcat [**?**], and so on.

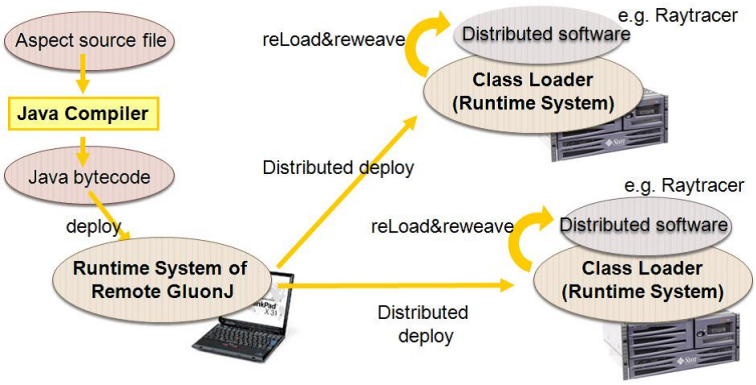

Figure 5.4. Architecture of hot deployment provided by the runtime system of Remote GluonJ

on top of the runtime system by Remote GluonJ. The developers of Remote GluonJ do not have to shutdown a working program since they apply new aspects to the program on its runtime system. Before the users start up their application on top of the runtime system provided by Remote GluonJ, they must put the classfiles on a directory specified by the runtime system. When the classfiles of aspects and classes on the specific directory are changed, all runtime systems on participating nodes automatically restart distributed software. When restarting distributed application, a runtime system on each node changes the behavior of classes at load time according to the context of the changed aspect.

The runtime system provided by Remote GluonJ observes classfiles on a specific directory at runtime. It checks if the last-modified time of the classfile is changed or not. If the time of the file is changed and the file is the classfile of an aspect, it gets the data of its aspect. It sends the participating runtime systems the data. The runtime system that received the data creates new classloader and makes the classloader reload the classfiles of a working application. It and its classloader edit the bytecode of the loading class at load time and then load the modified class like Figure 5.4.

5.5.4 Deadlock Avoidance

Any host can invoke a method on a remote object and receive a method invocation from a remote object in Remote GluonJ runtime. Therefore, a

remote method call from a host A to an other host B may cause another method call back from B to A. In this case, the latter method call must be handled by the same thread that requested the former method call on the host A. Otherwise, a deadlock may occur if the methods are synchronized ones.

In order to ensure the same thread executes all the methods called back, Remote GluonJ establishes a one-to-one communication channel between the thread executing a method on the host B and the thread executing a method that is handled the method on the host A. This communication channel is stored in a thread-local variable implemented with java.lang.ThreadLocal. The ThreadLocal class provides thread-local variables for the Java developers. These variables differ from their normal counterparts in that each thread that accesses one has its own, independently initialized copy of the variable.

A thread always uses the same channel for every remote method call and it waits for not only the result of the invocation but also another request of invocation from a remote thread sharing the same channel. A deadlock is avoided.

5.5.5 Distributed Garbage Collection

Remote GluonJ maintains a table of objects exported to a remote host. While there exists a proxy object on a remote host, the master object is recorded in that table so that it is not garbage collected. If all the proxy objects are garbage collected, then the master object is removed from the table. If there are no other references to the master object, then the master object is garbage collected.

The table of proxy objects for checking the equality between remote references is implemented with the weak reference mechanism of Java. An element of the table is a weak reference to a proxy object. Thus, the proxy object is garbage collected when the garbage collector determines that nothing except that table refers to the proxy object.

Currently, Remote GluonJ cannot collect all objects if remote references make cycles. Although several algorithms are known for dealing with distributed cycles, efficiently implementing those algorithms is not straightforward without modifying the JVM. For example, if using a distributed mark-sweep algorithm, we would need a mechanism for tracing object references. However, Java 's reflection API does not provide such a mechanism. We expect that weak references and object finalizers might

help to solve this problem but implementation details are still open.

5.5.6 Thread Pool

One simplistic model for building a RMI server program would be to create a new thread each time a request arrives and serivce the request in the new thread. This approach actually works fine for prototyping though, has significant disadvantages that would become apparent if you tried to deploy the RMI server program that worked this way. One of the disadvantages of the thread-per-request approach is that the overhead of creating a new thread for each request is significant, the server that created a new thread for each request would spend more time and consume more system resources creating and destroying threads than it would processing actual user requests. In addition to the overhead of creating and destroying threads, active threads consume system resources. Creating too many threads in one JVM can cause the system to run out of memory or thrash due to excessive memory consumption.

To resolve this problem, current Remote GluonJ tries to reduce the overhead of creating and destroying threads by using the thread pool [31]. We implement a thread pool that combined with a fixed group of worker threads. The thread pool uses wait() and notify to signal waiting threads that new work has arrived. The thread pool in Remote GluonJ meets the requirements for safely using notify(). The thread pool offers a solution to both the problem of thread life-cycle overhead and the problem of resource thrashing. By reusing threads for multiple tasks, the thread-creation overhead is spread over many tasks. As a bonus, because the thread already exists when a request arrives, the delay introduced by thread creation is eliminated. Thus, the request can be serviced immediately, rendering the application more responsive. Furthermore, by properly tuning the number of threads in the thread pool, you can prevent resource thrashing by forcing any requests in excess of a certain threshold to wait until a thread is available to process it.

5.6 Related work

Soares et al reported that they could use AspectJ for improving the modularity of their program written using Java RMI [71]. Without AspectJ, the program must include the code following the programming conven-

tions required by the Java RMI. AspectJ allows separation of that code from the rest into a distribution aspect. However, the ability of AspectJ is limited with respect to modularization for distributed programs and thus the resulting programs are often complicated and difficult to maintain. To address these complications, we propose remote pointcuts and the inter-type declaration as extended language constructs for distributed aspect-oriented programs.

Although Java RMI is the standard framework, several researchers have been proposing other systems such as cJVM [17], our own Addistant [51] and J-Orchestra [77]. These systems provide a single virtual machine image on several hosts connected through a network. They allow for the distributed execution of a program originally written as a non-distributed one, without code modification for the distribution. An alternative to the approach presented here might be to write a program in AspectJ and run it on these systems, which would appropriately translate local pointcuts into remote pointcuts at the implementation level. We did not take this approach since our target applications are for the unit testing of enterprise server software, and these programs are inherently designed and implemented as distributed software. Therefore, we do not have to translate such software to distributed software by, for example, using Addistant, except for the modules implemented as aspects. If we translate all the modules of such software, the unnecessary indirections due to the proxy objects would cause significant performance penalties, since such software has already included indirections for remote accesses. On the other hand, Remote GluonJ can be regarded as a system that translates only aspects to enable transparent remote accesses. Although Addistant allows the programmers to specify translation only for the classes generated by the AspectJ compiler from the aspects, the programmers must manually describe these specifications. Remote GluonJ provides better syntax so that these specifications can be simple or implicit within the language constructs.

Distribution is a well known crosscutting concern and several systems have been proposed to support such concerns. For example, the D language [46] allows the programmers to separately describe how a parameter is passed to a remote procedure. Such work has explores new crosscutting concerns in distributed programs whereas our work explore general-purpose language constructs for distributed aspect-oriented programs. The goal of our work is to develop language constructs so that programs written in an AspectJ-like language can be simple and easy to

maintain.

JAC [62, 57] is a powerful framework for dynamic AOP in Java. Unlike other lanugages such as AspectJ, JAC does not require any langugae extensions to Java. An aspect of JAC is implemented by a set of aspect objects. JAC also supports Java API that easily implements crosscutting concerns in distributed systems such as the codes changing consistency protocol on a set of replications and implementing load-balancing for developers. But, using JAC, even if developers will separate the crosscutting concerns during unit testing, complicated network processing is not necessarily solved. The significant difference JAC and Remote GluonJ is that Remote GluonJ provide the remote pointcut.

DADO (Distributed Adaplets for Distributed Objects) [82] provides a CORBA-like programming model, which comprises several languages, tools, and runtime environment, to support crosscuting concerns in distributed heterogeneous systems. This programming model enables the developers to separate crosscutting implementation that arised in application components on both client and server side such as security, caching. In particular, the DADO programming model has two languages. One of these languages, DADO deployment language, is based on AspectJ and specifies how a QoS feature interacts with an underlying application. However these languages allow modeling the communications between client and server side, don't support remote pointcuts provided by Remote GluonJ.

5.7 Summary

This chapter presented Remote GluonJ, which provides remote pointcut as a new language construct for distributed AOP. A remote pointcut is a function for identifying join points in the execution of a program running on a remote host. It can simplify the description of aspects with respect to network processing if the aspects implement a crosscutting concern spanning over multiple hosts. To illustrate this situation, this chapter used the example of a program written in Remote GluonJ for distributed unit testing. The remote pointcut is a crucial language construct for distributed AOP, corresponding to remote method invocation (RMI) for distributed object-oriented programming.

Chapter 6

Conclusion

This thesis has presented an aspect-oriented programming (AOP) language for agile software development named GluonJ. This chapter concludes the thesis with a summary of our contribution.

First this thesis explained four features of an ideal language for agile software development and showed that AspectJ is the most ideal language for agile software development in existing programming languages. The ideal features are as following.

(A) Separation of features

(B) Without editing an existing program

(C) Concurrent development

(D) A statically typed language

AspectJ is one of general-purpose AOP languages for Java and provides language constructs for pointcut-advice and inter-type declarations. By using pointcut-advice and inter-type declarations in AspectJ satisfy the features (B) and (D). Although AspectJ also satisfies the features (A) and (C) to a certain degree, it is not good enough.

Our contribution is that we proposed two new language constructs for AOP, named dynamic refinement and remote pointcut. Dynamic refinement is an AOP language construct for concurrent development. It allows software engineers to write extensions to existing classes according to dynamic contexts. Remote pointcut is an AOP language construct for distributed computing and allows separating a distributed feature from the rest of the program as a simple aspect. We also developed GluonJ, which is an AspectJ-like AOP language for Java. It provides these language constructs and satisfies the ideal features not only (B) and (D) but also (A) and (C). We developed GluonJ that provides these language constructs.

6.1 Dynamic refinement

We proposed dynamic refinement, which allows software engineers to dynamically refine the definition of an existing class. By using dynamic refinement in GluonJ, software engineers can redefine existing methods and append new methods, fields, and interfaces to an existing class according to control-flow. Since these changes are described in a separate component (or module), this language mechanism is useful for separation of concerns.

Dynamic refinement is effectively used for development of web applications. The programs of most web applications have session objects and a mechanism for managing these objects. Dynamic contexts of a web application are registered and centralized in a session object. Software engineers of web applications use these session objects and describe the codes for switching processings according to sessions by hand. By using dynamic refinement in GluonJ, they do not have to write such codes. GluonJ automatically enables switching the the processings declared in refinements by control-flow (dynamic scoping). Moreover GluonJ allows them to group extensions to existing classes related to a same session as refinements.

GluonJ that provides dynamic refinement satisfies the feature (C) of an ideal language for agile software development. Dynamic refinement allows software engineers to separate newly appended functions from the rest of the program and other functions as aspects. A software engineer can develop his function without consideration of different functions that other engineers develop at same iteration. Also it is easy to merge de-

veloped functions into an existing program. Unlike cflow pointcut in AspectJ, dynamic refinement in GluonJ provides a mechanism for grouping multiple redefined methods according to a same dynamic context.

The concept of dynamic refinement is similar to the mechanism of dynamic scoping. Generally speaking, a language that has dynamic scoping is not easy-to-use. Such a language decreases the readability and understandability of its program. However, when software engineers, in parallel, improve a software product on agile software development, dynamic scoping is often effective. Thus GluonJ exploits dynamic scoping.

GluonJ allows software engineers to dynamically refine an existing class during runtime to a certain degree. It allows applying and removing refinement to/from a class on demand. Naively designed dynamic refinement may allow a call to a method that has not been appended yet or that has been already removed and then it may cause a runtime type error. However, a GluonJ program never causes such a runtime type error as a NoSuchMethodException although it may fail an explicit type cast and throw a ClassCastException. To guarantee this property, GluonJ does not use a custom type checker. It exploits the type checking by a normal Java compiler and the verification by the custom class loader of GluonJ.

6.2 Remote pointcut

We proposed remote pointcut, which is new AOP language constructs for distributed computing. A remote pointcut is a function for designating join points in the execution of a program running on a remote host. Although a pointcut in AspectJ identifies execution points on the local host, a remote pointcut can identify them on a remote host. In other words, when the thread of control reaches the join points identified by a remote pointcut, the advice body associated with that remote pointcut is executed on a remote host different from the one where those join points occur. Remote pointcuts are analogous to remote method calls, which invoke the execution of a method body on a remote host.

GluonJ that provides remote pointcut satisfies the feature (A) of an ideal language for agile software development in Section 2.2. This language construct can simplify the code of a component implementing a feature in distributed software as an aspect. AspectJ is a useful programming language is a useful programming language for developing dis-

tributed software. However, even if a feature can be implemented as a single aspect at the code level, it might need to be deployed on different hosts and it would therefore consist of several sub-components or sub-processes running on each host. Remote pointcut enables implementing such a feature as simple, non-distirbuted component without concerns about network processing.

The aspect weaving in GluonJ is performed at load time on each participating host. The normal Java classes on each participating host must be loaded by the class loader provided by GluonJ. This class loader weaves aspects and classes on the fly. The compiled aspects are stored in the aspect server. The parts of the compiled code except for the advice bodies are automatically distributed by the aspect server to each host, so the latest aspects can be woven when the classes are loaded. The software engineers of GluonJ do not have to manually deploy the compiled aspects to every host.

AOP are useful for agile software development. However, existing AOP languages, which include GluonJ, are not good enough for agile software development.

Existing AOP languages do not often allow implementing a new function as an aspect. This chapter proposes to use AOP languages for agile software development and then to implement a newly appended function as an aspect. If this addition of an aspect is iterated in software development, a developer must often append a new function to an existing aspect that was developed before by an aspect. Existing AOP languages such as AspectJ make it difficult to extend an existing aspect by an aspect. This is because an advice in those languages is an anonymous method. Thus they cannot appropriately specify the execution or call of an advice as join points. To avoid this problem, before developers append a new function to an existing program in an iteration, they should refactor the existing program that includes several aspects every several iterations. After they change an existing program into a program without aspects, they should append a new function to the refactored program.

Existing AOP languages easily enable deleting the implementations of functions (aspects) that have been developed by the previous iteration only. In existing AOP languages, the deletion of an aspect on which other aspects are depended requires care. For example, suppose that an intertype declaration is declared in an aspect A1 and the execution of the intertype declaration is specified by another aspect A2 as join points. If a developer removes aspect A1 from the rest of the program without

consideration of aspect A2, the aspect A2 is not effective. Thus, when a developer removes an aspect from the rest of the program, he should check the dependency relation between its aspect and other aspects. Note that, he would easily delete an aspect only that was developed by the previous iteration.

Bibliography

[1] *Agile Alliance*. URI http://www.agilealliance.com/.

[2] *aspectj project — aspectj crosscutting objects for better modularity*. URL http://www.eclipse.org/aspectj/.

[3] *AspectWerkz - Plain Java AOP*. URI http://aspectwerkz.codehaus.org/.

[4] *CaesarJ Project*. URL http://caesarj.org/.

[5] *eclipse*. URL http://www.eclipse.org/.

[6] *JBoss Aspect Oriented Programming (AOP)*. URL http://labs.jboss.com/portal/jbossaop.

[7] *Jikes Research Virtual Machine*. Online publishing, URI http://jikesrvm.sourceforge.net/.

[8] *Manifesto for Agile Software Development*. URI http://agilemanifesto.org/.

[9] *NetBeans*. URL http://www.netbeans.org.

[10] *Rational Unified Process*. URI http://www-306.ibm.com/software/awdtools/rup/?S_TACT=105AGY59&S_CMP=WIK%I&ca=dtl-08rupsite.

[11] *The AspectJ 5 Development Kit Developer's Notebook*. URL http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html.

[12] *The Scala language home page.* URL `http://scala.epfl.ch/`.

[13] *GluonJ Home Page.* URL `http://www.csg.is.titech.ac.jp/projects/gluonj/`, 2006.

[14] Alexandre Bergel and Stephane Ducasse and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of the Joint Modular Languages Conference 2003 (JMLC '03)*, volume 2789 of *Lecture Notes in Computer Science*, pages 122–131. Springer-Verlag, 2003.

[15] Sven Apel and Don Batory. When to use features and aspects?: a case study. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06)*, pages 59–68, New York, NY, USA, 2006. ACM.

[16] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An Overview of CaesarJ. In Awais Rashid and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development I*, Lecture Notes in Computer Science, pages 135–173. Springer, 2006.

[17] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: A single system image of a JVM on a cluster. In *Proceedings of International Conference on Parallel Processing (ICPP '99)*, pages 4–11, 1999.

[18] asset.com.

[19] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*, pages 177–189, New York, NY, USA, October 2005. ACM Press.

[20] Lodewijk Bergmans and Mehmet Aksits. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, 2001.

[21] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04)*, pages 83–92, March 2004.

[22] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP '90)*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[23] Shigeru Chiba. Load-Time Structural Reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP '00)*, pages 313–336, London, UK, June 2000. Springer-Verlag.

[24] Shigeru Chiba and Rei Ishikawa. Aspect-Oriented Programming Beyond Dependency Injection. In Andrew P. Black, editor, *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP '05)*, volume 3586 of *Lecture Notes in Computer Science*, pages 121–143. Springer Berlin/Heidelberg, 2005.

[25] Shigeru Chiba and Muga Nishizawa. An Easy-to-Use Tookkit for Efficient Java Bytecode Translators. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE '03)*, volume 2830 of *Lecture Notes in Computer Science*, pages 364–376, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[26] W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Department of Computer Science, Brown University, Providence, RI, May 1989.

[27] Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-oriented Programming: an Overview of ContextL. In *Proceedings of the 2005 Conference on Dynamic Languages Symposium (DLS '05)*, pages 1–10, New York, NY, USA, 2005. ACM Press.

[28] Pascal Costanza, Robert Hirschfeld, and Wolfgang De Meuter. Efficient Layer Activation for Switching Context-dependent Behavior. volume 4228 of *Lecture Notes in Computer Science*, pages 84–103. Springer, September 2006.

[29] Curtis Clifton and Gary T. Leavens and Craig Chambers and Todd Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings of the 15th ACM SIGPLAN*

conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00), pages 130–145, New York, NY, USA, 2000. ACM Press.

[30] Mxx Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In *Workshop on Engineering Complex Object-Oriented Systems for Evolutions, Proceedings of Object-Oriented Programming Softwares, Languages and Applications (OOPSLA '01)*, October 2001.

[31] Doug Lea. *Concurrent Programming Second Edition Design Principles and Patterns*, chapter 4. Addison Wesley Java Series, 2000.

[32] Marc Fleury and Francisco Reverbel. The JBoss extensible server. In Markus Endler and Douglas Schmidt, editors, *Proceedings of ACM/IFIP/USENIX 4th International Middleware Conference (Middleware '03)*, volume 2672 of *Lecture Notes in Computer Science*, pages 344–373. Springer-Verlag, 2003.

[33] Gregor Kiczales and John Lamping and Anurag Menhdhekar and Chris Maeda and Cristina Lopes and Jean-Marc Loingtier and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the eleven European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[34] Mills H., Dyer M., and Linger R. Cleanroom software engineering. In *IEEE Software 4, 5*, pages 19–25, September 1987.

[35] JBoss Inc., URL `http://www.jobss.org/`. *JBoss 4.0.2*, 2004.

[36] John Corwin and David F. Bacon and David Grove and Chet Murthy. MJ: a rational module system for Java and its applications. *SIGPLAN Not.*, 38(11):241–254, 2003.

[37] Kent Beck. *Extreme Programming Explained: Embrace Change*, chapter 4. Addison-Wesley Professional, October 1999.

[38] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, volume 2072, pages 327–355. Springer-Verlag, 2001.

[39] Michael Kircher, Prashant Jain, and Angelo Corsaro. *XP + AOP = Better Software?* URI `http://agilealliance.com/system/article/file/882/`.

[40] Klaus Ostermann. Implementing Reusable Collaborations with Delegation Layers. In *First Workshop on Language Mechanisms for Programming Software Components at OOPSLA'01*, October 2001.

[41] Klaus Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02)*, pages 89–110, Malaga, Spain, 2002. Springer.

[42] Gunter Kniesel. Type-Safe Delegation for Run-Time Component Adaptation. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*, pages 351–366, Lisbon, Portugal, June 1999. Springer-Verlag.

[43] Kresten Krab Thorup. Genericity in Java with Virtual Types. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the eleven European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241, pages 444–471, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[44] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '98)*, pages 36–44, New York, NY, USA, 1998. ACM Press.

[45] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPLSA '86)*, pages 214–223, New York, NY, USA, 1986. ACM Press.

[46] Cristina Lopes. *D: A Language Framework for Distributed Programming.* PhD thesis, College of Computer Science, Northeastern University, Dec. 1997.

[47] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA '89)*, pages 397–406, New York, NY, USA, 1989. ACM Press.

[48] Ole Lehrmann Madsen, Birger Mø-Pedersen, and Kristen Nygaard. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[49] Matthias Zenger. Evolving Software with Extensible Modules. In *International Workshop on Unanticipated Software Evolution*, Malaga, Spain, June 2002.

[50] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD '03)*, pages 90–99, New York, NY, USA, March 2003. ACM Press.

[51] Michiaki Tatsubori and Toshiyuki Sasaki and Shigeru Chiba and Kozo Itano. A Bytecode Translator for Distributed Execution of Legacy Java Software. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 236–255. Springer-Verlag, 2001.

[52] Nathanael Scharly and Ste phane Ducasse and Oscar Nierstrasz and Andrew Black. Traits: Composable Units of Behavior. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '03)*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274, New York, NY, USA, July 2003. Springer.

[53] Muga Nishizawa and Shigeru Chiba. A small extension to java for class refinement. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (ACM SAC '08)*. ACM, 2008.

[54] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut: a language construct for distributed AOP. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04)*, pages 7–15, New York, NY, USA, March 2004. ACM Press.

[55] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '04)*, pages 99–115, New York, NY, USA, October 2004. ACM Press.

[56] Object Mentor, URI `http://www.junit.org/index.htm`. *JUnit.org*, 2001.

[57] ObjectWeb Consortium, Online publishing, URI `http://jac.objectweb.org/index.html`.

[58] Doug Orleans and Karl J. Lieberherr. Dj: Dynamic adaptive programming in java. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Proceedings of the 3rd International Conference of Metalevel Architectures and Separation of Crosscutting Concerns (Reflection '01)*, volume 2192 of *Lecture Notes in Computer Science*, pages 73–80. Springer-Verlag, 2001.

[59] Harold Ossher and Peri Tarr. Hyper/J: Multi-dimensional Separation of Concerns for Java. In *Proceedings of the 22nd international conference on Software engineering (ICSE '00)*, pages 734–737, 2000.

[60] Penaud Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier, and Laurent Martelli. Jac: An aspect-based distributed dynamic framework. In *Aspect-Oriented Software Development 2003 (AOSD 2003) Tutorial*, 2003.

[61] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. JAC : An Aspect-based Distributed Dynamic Framework. *Software Practise and Experience (SPE)*, 34(12):1119–1148, October 2004.

[62] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. Jac: A flexible solution for aspect-oriented programming in java. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Proceedings of the 3rd International Conference of Metalevel Architectures and Separation of Crosscutting Concerns (Reflection '01)*, volume 2192 of *Lecture Notes in Computer Science*, pages 1–24. Springer-Verlag, 2001.

[63] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 100–109, New York, NY, USA, March 2003. ACM.

[64] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD '02)*, pages 141–147, New York, NY, USA, March 2002. ACM Press.

[65] Hridesh Rajan and Kevin J. Sullivan. Classpects: Unifying Aspect- and Object-Oriented Language Design, booktitle =.

[66] Winston. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of IEEE WESCON 26*, pages 1–9, August 1970.

[67] Winston. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 328–338, 1987.

[68] Yoshiki Sato and Shigeru Chiba. Loosely-Separated "Sister" Namespaces in Java. In Andrew P. Black, editor, *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP '05)*, volume 3586 of *Lecture Notes in Computer Science*, pages 49–70. Springer Berlin/Heidelberg, 2005.

[69] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A selective, just-in-time aspect weaver. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE '03)*, volume 2830 of *Lecture Notes in Computer Science*, pages 189–208, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[70] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[71] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with aspectj. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, pages 174–190, New York, NY, USA, 2002. ACM Press.

[72] Sun Microsystems, Online publishing, URI `http://java.sun.com/j2se/1.5.0/docs/guide/jpda/index.html`. *Java Platform Debugger Architecture*.

[73] Sun Microsystems, URL `http://java.sun.com/products/jdk/rmi/`. *Java RMI*.

[74] Sun Microsystems, URL `http://java.sun.com/j2se/1.4.1/docs/guide/rmi/socketfactory/`. *Using a Custom RMI Socket Factory*, 1997.

[75] Davy Suvee, Wim Vanderperren, and Viviane Jonckers. JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development. In *Proceedings of the 2nd International Conference of Aspect-Oriented Software Development (AOSD '03)*, pages 21–29. ACM Press, March 2003.

[76] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering (ICSE '99)*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[77] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic java application partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02)*, pages 178–204, Malaga, Spain, 2002. Springer.

[78] Walter Maner, URL `http://csweb.cs.bgsu.edu/maner/domains/RAD.htm`. *RAPID APPLICATION DEVELOPMENT*, 1997.

[79] Alessandro Warth, Milan Stanojevic, and Todd Millstein. Statically Scoped Object Adaptation with Expanders. In *Proceedings of the 21th annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '06)*, New York, NY, USA, October 2006. ACM Press.

[80] Gilani Wasif and Spinczyk Olaf. Dynamic Aspect Weaver Family for Family-based Adaptable Systems. In *Proceedings of Net.ObjectDays 2005*, pages 94–109. Lecture Notes in Computer Science, September 2005.

[81] J. C. Wichman. *The Development of a Preprocessor to Facilitate Composition Filters in the Java Language*. PhD thesis, Dept. of Computer Science, University of Twente, December 1999.

[82] Eric Wohlstadter, Stoney Jackson, and Premkumar Devanbu. Dado: Enhancing middleware to support cross-cutting features in distributed, heterogeneous systems. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 174–186. ACM Press, May 2003.

[83] Yuuji Ichisugi and Akira Tanaka. Difference-Based Modules: A Class-Independent Module Mechanism. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02)*, pages 62–88, London, UK, 2002. Springer-Verlag.