An AOP-based Performance Evaluation Framework for UML models

Dong Kwan Kim and Shawn Bohner
Virginia Tech, Department of Computer Science
Blacksburg, VA 24061, USA
Email: {ikek70, sbohner}@cs.vt.edu

Abstract

Performance is a key aspect of non-functional software requirements. While performance cross-cuts much of the software functionality, it is frequently difficult to express in traditional software development representations. In this paper we propose a framework for evaluating software performance using aspect-oriented programming (AOP) and examine its strengths and limitations. The framework provides a mechanism for supporting software performance evaluation prior to final software release. AOP is a promising software engineering technique for expressing cross-cutting characteristics of software systems. We consider software performance as a cross-cutting concern since it is not confined only a few modules, but often spread over multiple functional and non-functional elements. A key strength of our framework is the use of a code instrumentation mechanism of AOP – AspectJ code for performance analysis is separated from Java code for functional requirements. Java code is executable regardless of AspectJ code and can be woven together with AspectJ code when performance is evaluated. Our performance evaluation modules, written in AspectJ are semi-automatically or automatically generated from the UML [1] models with annotated performance profiles. The AspectJ code generator facilitates performance evaluation by allowing performance requirements that have been specified in UML models to be analyzed. The UML diagrams can then be improved by reflecting the feedback from the results of the performance analysis.

Keywords

Performance Modeling, Aspect-Oriented Programming (AOP), Software performance, Unified Modeling Language (UML)

1. Introduction

Generally speaking, software requirements are partitioned into functional and non-functional requirements. Functional requirements are associated with specific functions, tasks or behaviors the system must support, while non-functional requirements are often constraints or elaborations on various attributes of these functions or tasks. The non-functional requirements that serve as software quality attributes include accuracy, modifiability, security and performance [2]. A final product could be worthless unless it satisfies the non-functional requirements. In this work we are concerned about how well we can express and evaluate software performance among non-functional requirements.

With performance often cross-cutting requirements, it makes sense to use Aspect-Oriented Programming (AOP) to represent this aspect of requirements, since that is the purported strength of this technique. We define and localize performance as an aspect of the software using AOP. This brings together performance requirements over the software components of the system into an artifact that can be instrumented for performance analysis, yet remain consistent with the relevant models. We believe that this could increase the maintainability and understandability of the system.

It is important to identify software performance problems early in the development life cycle. However, this often involves sophisticated performance models that can mathematically represent performance aspects of the software system. We introduce an alternative AOP-based Performance Evaluation Framework (APEF) to aid software engineers in evaluating the performance of software systems. It is based on AOP and generates AspectJ code that is employed to test the performance requirements.

While APEF, in its current form, does not provide for performance prediction, it does provide for the evaluation of performance requirements. The testing components for the performance evaluation are generated from UML models and instrumented in the code. These are then used to ensure that the performance aspects are tested. Based on our preliminary research, we anticipate improved performance verification as well as development cost and time savings for performance evaluation using APEF.

Following this brief introduction, Section 2 outlines some background information on UML performance profiles, AOP, and AspectJ. Section 3 discusses related work in model-based performance modeling/prediction, representing AOP in UML, and performance profiling using AspectJ. In Sections 4-5, our proposed framework

is discussed along with examples of how it is applied. We then conclude the paper with key observations.

2. Background

2.1. UML Performance Profile

The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems [3] that provides multiple diagrams for software developers to analyze and design structural or dynamical aspects of object-oriented software systems.

The Object Management Group (OMG) has published UML profiles for schedulability, performance, and time specification [4] that defines a set of quantitative performance annotations to be added to a given UML model. It facilitates capturing performance requirements, associating performance-related Quality of Service (QoS) characteristics, specifying execution parameters, and presenting performance results computed by modeling tools or found in testing [4] [5]. The profile allows UML diagrams to be annotated with performance information such as workload, scenario, step, and resource. In section 4, an example of sequence diagram with performance profile will be presented. Figure 1 depicts a general performance model that identifies the basic abstractions and relationships.

2.2. Aspect Oriented Programming (AOP)

AOP is a programming paradigm that promotes the principle of separation of concerns to enable programmers to focus on aspects [6]. It is a promising technology that goes beyond object-oriented programming to support the entire software development life cycle from requirements analysis to testing. For example, aspect-oriented requirements engineering is a discipline to study how well aspects can be captured from requirements – testing techniques also benefit here. Various performance concerns transcend multiple classes when developing software systems. Frequently, they are non-functional requirements such as logging, persistence, concurrency, and security. The major advantage of AOP is to localize those cross-cutting concerns as aspects. As scattered code is located in one artifact, entire code is more concise as well as maintainability and reusability are improved.

2.3. AspectJ

AspectJ [6] [7] is an aspect-oriented extension to the Java programming language that provides convenient constructs for supporting separation of concern concepts like *join point*, *pointcut*, *advice*, and *weaving*. *Join point* is a well-defined point in the program flow. It consists of method calls, method executions, object instantiations, constructors executions, field references and handler executions. A *pointcut* picks out certain join points and values at those points [8] [9]. Boolean operations or wildcards allow programmers to flexibly pick key join points. *Advice* is executed when a join point is reached. It extends or modifies the original joint point function by defining additional behaviors. The code fragment can be executed before, after, or around a given pointcut [10]. Pointcut and advice are useful constructs to dynamically affect execution flow of Java program.

The main work of AspectJ implementation is to ensure that aspect and non-aspect code (normally, Java code) run together in a properly coordinated fashion. This coordination process is called aspect weaving and involves making sure that applicable advice runs at the appropriate join points [7].

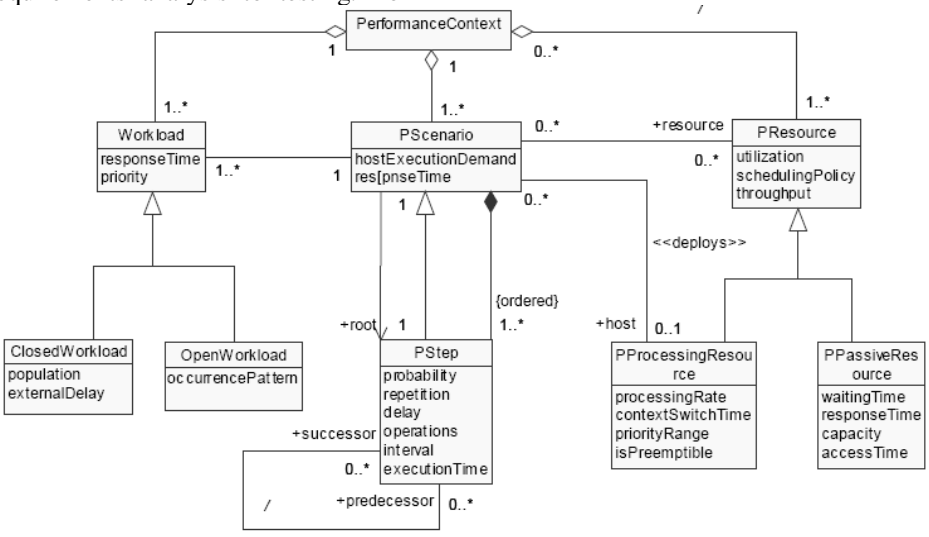

Figure 1 – Performance analysis domain model in UML [4]

3. Related Work

Over the past few decades, performance analysis has received significant attention to support software design. Leading approaches include queueing network-based optimization, process-algebra-based, Petri Net-based, simulation method, and stochastic process-based approaches [11]. Most of the approaches provide their own performance models that can be created during early design phase. Their primary goal is to identify software having unacceptable performance.

In the design phase, it is often necessary to simulate the performance models to predict performance. Learning and applying formal performance models and corresponding simulation software is also difficult and time consuming. Hence, it is practiced infrequently.

More recently, research based on the UML performance domain model has emerged where a performance model can be generated from UML models using performance profiles. The performance analysis domain model is transformed into the eXtensible Markup Language (XML), an intermediate format that is widely accepted across various applications [3]. The intermediate file schema is defined to represent all the essential performance information from the original UML model. In [12], Arief and Speirs introduce a tool that allows system developers to design a new system using the UML class and sequence diagram notations. The tool automatically generates simulation programs from those diagrams along with some random and statistics information. Park and Kang [13] show how AspectJ code can be constructed from UML-based performance models and used for simulating performance prior to coding the software.

Many research papers have introduced UML extensions in order to specify AOP concepts such as join points, advice, and pointcuts. Aldawud et al [14] presents an Aspect-Oriented Software Development (AOSD) UML Profile that supports specifying, visualizing, and documenting the artifacts of software systems. It uses UML extension mechanisms to be smoothly integrated into existing CASE tools that support UML. Basch and Sanchez [15] propose two new modeling elements to incorporate aspects into UML. One is a join point at which aspects cross-cut components in the main program, and the other is an aspect itself. In this approach, the aspects are encapsulated into distinct UML packages to achieve separation of concerns. Stein et al [16] focus on how to represent join points using UML classifiers and links. They examine join points in AspectJ, Composition Filters, Adaptive Programming, and Hyper/J.

Ho et al [17] propose a UML All pUrpose Transformer (UMLAUT) framework as methodological support for building and manipulating UML models with aspects. This allows software engineers to program the weaving of aspects at the level of the UML metamodel.

Bodkin [18] addresses how to use AspectJ in order to monitor performance of applications. He introduces a basic aspect-oriented performance monitoring system supporting Servlets monitoring, database request monitoring, Servlet request tracking, and Java Database Connectivity (JDBC) monitoring. It is a guideline to implement aspect-oriented performance frameworks. Pearce et al [19] introduce an experimental aspect-oriented Java profiler, called DJProf that employs AspectJ to insert the necessary instrumentation for profiling and can be used to profile Java programs without modification and does not require users to have any knowledge of AspectJ. DJProf also supports several different modes of profiling e.g. heap usage, object lifetime, wasted time, and time spent.

Many of these efforts have provided valuable insights and motivation to develop the APEF that this paper reports on in the next section.

4. Proposed AOP-based Performance Evaluation Framework

APEF supports the approach which is illustrated in Figure 2 (b). Compared with the traditional approach in Figure 2 (a) the advantages of APEF are to localize performance requirements as aspects (i.e. isolate them from business logic of software systems) and to generate AspectJ code for easily evaluating the performance requirements. Both approaches in Figure 2 take UML models with performance profile. However, the implementation part of our approach involves both Java and AspectJ code (unlike the traditional approach that employs only one implementation language. It is reasonable to separate implementation into Java and AspectJ since we model functional specifications and performance requirements. Functional specifications are implemented in Java and performance evaluation specifications are constructed in AspectJ.

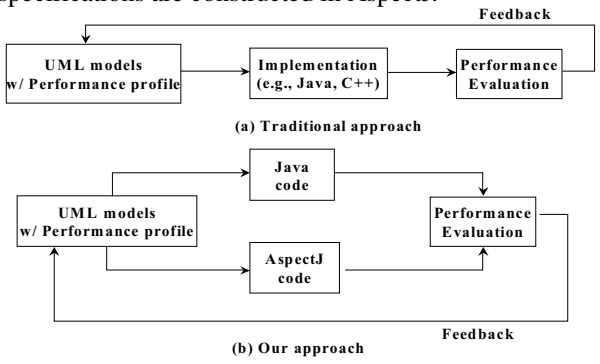

Figure 2 – Comparing the Approaches

The traditional approach could make software systems more complicated because code for performance evaluation scatters over multiple components for functionalities. It could also result in bad maintainability of software systems. As mentioned in previous sections,

APEF generates AspectJ code to allow users easily to evaluate performance. The AspectJ code can be generated from UML models annotated with performance profiles. Therefore, the input of APEF should be UML models. Model Manager takes UML models and then creates performance model which contains performance requirements. Code generator generates AspectJ and Java code based on UML and performance models. APEF finally produces results of performance analysis after together executing AspectJ and Java code. Figure 3 illustrates a structure of APEF that will be explained in the following subsections.

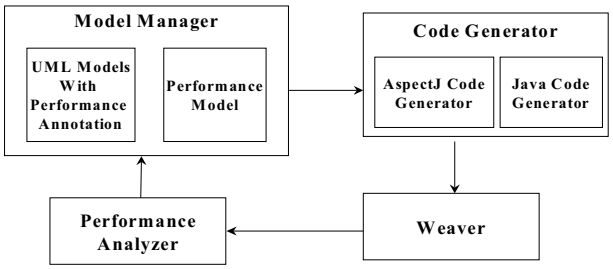

Figure 3 – Structure of APEF

APEF consists of the following modules.

- **Model Manager** – manages UML and performance models. It takes UML models with performance profiles as inputs of APEF and extracts performance information from them. Mainly class and sequence diagrams are considered to be explored to obtain the performance constraints or requirements. Other diagrams such as activity and deployment diagrams can also be considered so that a more accurate performance model can be created. Rules for generating performance model have to be predefined formally.
- **Code Generator** – is a main key of APEF that generates the AspectJ and Java code. APEF automatically or semi-automatically generates AspectJ code from performance models and AspectJ code templates. Only skeleton code for Java classes can be generated from class diagrams. Detailed implementations for those Java classes need to be manually written.
- **Weaver** – combines Java classes with performance aspects so that both are executed together. During compile, loading, or run time performance aspects instrument Java classes according to their pointcuts and advice in order to evaluate performance of software systems. APEF is based on AspectJ that supports compile-time and load-time weaving.
- **Performance Analyzer** – analyzes results of performance evaluation and produces feedback which may be applied into either UML or performance models. It needs to have components that enable software developers to collect information such as response time, throughput, average service time, or number of processed jobs. Here is where one can change some of performance constraints in order to improve performance of software systems under development.

Figure 4 illustrates how major four components of APEF works together. Two types of performance requirements are considered: General Performance Requirement (GPR) and Implementation-specific Performance Requirement (IPR). GPRs as implementation-independent requirements can be extracted in software analysis or design phase and they are specified and represented on the performance analysis domain model in UML. IPRs are dependent on software execution environments and resources, for example, CPU or heap usage. Both performance requirements can be modulized as aspects. GPRs are analyzed and specified by using UML models.

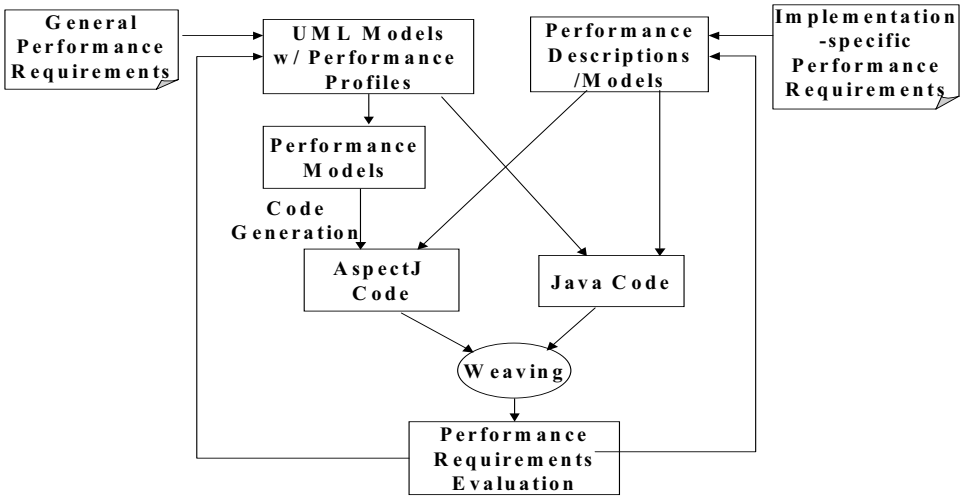

Figure 4 – Overall Flow of APEF

Many CASE tools provide for annotating UML diagrams with performance profiles. These tools usually also allow diagrams to be saved as XML Metadata Interchange (XMI) format [20] so that they can be exchangeable with other tools. Performance models can be created from such a XMI file containing GPRs. They are based on performance analysis domain model in UML and are represented by XML format. AspectJ code can be generated from those performance models. Code generator needs to parse them to extract performance information that is required to generate code. We do not discuss in this paper Java code generation since many existing UML tools support it.

Figure 5 depicts detailed procedures for generating AspectJ code. Code generation consists of design and implementation phases. Performance requirements are outputs of design phase and specify which aspects should be generated for evaluation. Those aspects will be used to evaluate the performance requirements. Performance requirements can be obtained from class and sequence diagrams. Those diagrams should contain performance profiles that are defined on performance domain analysis model. Another major component for code generation is code templates. It is common code for concrete performance aspects and is implemented based on performance requirements defined on performance analysis domain model.

Let's suppose a performance aspect for response time. Common parts of the aspect for response time are placed in the code template and specific ones come from performance requirements. AspectJ code generator constructs an executable AspectJ model from code templates and performance requirements. IPRs are specified by additional performance descriptions or models. Both GPRs and IPRs are necessary for code generator to generate AspectJ and Java code.

At the weaving step, AspectJ code is inserted at the appropriate spot of Java code or replaces behaviors of Java code. Evaluation results regarding performance requirements can be reflected into UML and performance models.

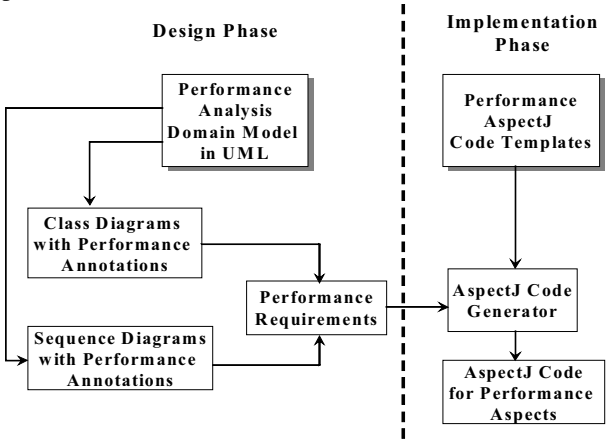

Figure 5 – Flow of AspectJ code generation

5. Experience Through Examples

To gain experience with the APEF, three examples (response time, population, and throughput) are examined to demonstrate how performance requirements are represented by aspects. A Web-based video-streaming application from OMG [4] is used. A user identifies and requests a video on a video server over the Internet. Figure 6 illustrates a sequence diagram with performance annotations for video request and display scenario in web video application. The following three examples show AspectJ code for performance requirements in Figure 6.

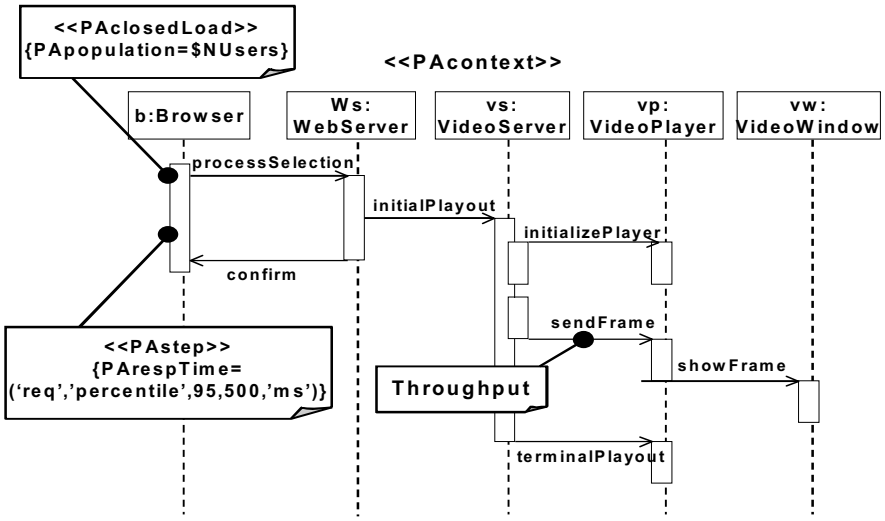

Figure 6 – Sequence Diagram with Performance Annotation [4]

5.1. An Aspect for Response Time

You can see a requirement for response time in Figure 6 illustrates a response time requirement and indicates a probability for the response time that the delay in receiving the conformation will not take longer than half a second in 95% of the cases:

Probability(Confirmation delay > 500ms) < 0.05

Before making AspectJ code for the response time, let's examine a class diagram for response time aspect. Figure 7 shows a class diagram having `ResponseTimeAJ` aspect and `WebServer` class. If the elapsed time for `processSelection` method exceeds 500 milliseconds the response time constraint isn't satisfied. In order to see if the constraint is satisfied, the execution time of `processSelection` needs to be measured. In Figure 7 `ResponseTimeAJ` aspect has one pointcut and two pieces of advice. Selected join point is a `processSelection` method of `WebServer`. `Before` advice measures a system time before the method is executed. `After` advice measures a system time right after it finishes. The difference between measured times is the execution time of the `processSelection` method. If it is less than 500 milliseconds the given response time condition is satisfied. Source code for the `ResponseTimeAJ` can be generated from the performance aspect code templates. The necessary information for the code generation comes from sequence and class diagrams that describe participating objects, methods, and performance profiles. Table 1 shows the information that is necessary to generate aspect code in this example. Performance requirements should be provided. A participating object and its method with parameters are used to generate a pointcut. Advice information depending on the pointcut needs also to be provided.

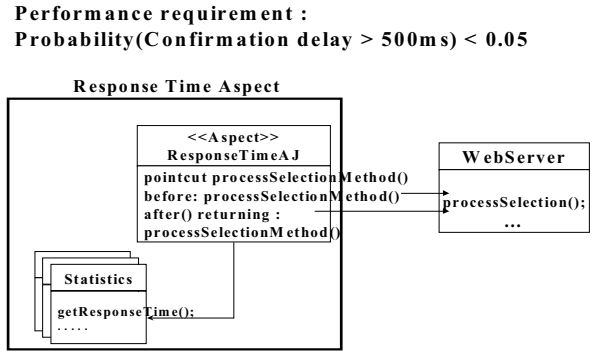

Figure 7 – Class Diagram with Response Time Aspect

Table 1 – Performance Data for Code Generation

Parameter names	Parameter values
Performance requirement	Probability(Confirmation delay > 500ms) < 0.05
Participants	WebServer
Method name	processSelection
Parameter types of the method	None
Number of parameters	0
Return type of the method	Void
Pointcut	processSelectionMethod()
Advice	Before: processSelectionMethod () after() returning : processSelectionMethod ()

Figure 8 gives us `ResponseTimeAJ` aspect which is generated from code templates. Before `WebServer.processSelection` is executed `before` advice runs and after it is finished `after` advice runs. The elapsed time is calculated in the body of `after` advice and if the response time exceeds 500 milliseconds a message would display on the standard output.

```
public aspect ResponseTimeAJ{
  long begin,end,rTime;
   pointcut processSelectionMethod():execution(void
   WebServer.processSelectionMovie());
       before() : processSelectionMethod() {
            begin=System.currentTimeMillis();
       }
       after() returning : processSelectionMethod() {
             end=System.currentTimeMillis();
             rTime=end-begin;
        if(rTime>500)
             System.out.println("The confirmation delay has
                                    exceeded 500 ms.");
       }
}
public class WebServer{
     public void processSelection(){
             //do something
     }
}
```

Figure 8 – Aspect for Response Time: ResponseTimeAJ

5.2. An Aspect for Population

A second example of performance requirement is population. Population is the size of the workload i.e. number of system users [4]. The example shows how population of the web video application can be implemented as an aspect. In Figure 6,

PApopulation=$NUsers indicates the population that is determined by an input value, NUsers. Let's suppose the $NUsers is 1,000 in this example. It is meant that the web video application allows only 1,000 users on the server at a time. We assume the web video application is constructed in a client-server architecture. A web server on the server side creates and assigns a thread for each user when a user visits a web page. Created threads are managed in a thread pool (i.e. they can be put into the pool or come out of the pool). The number of threads should not exceed 1,000 because the server allows only 1,000 users in this example.

```
public aspect PopulationAJ {
  static int population=1000;
  static int activeUser=0;
  UserPool pool = new UserPool();

  pointcut userConnection(Runnable runnable)
  : call(Thread.new(Runnable)) && args(runnable);
  Thread around(Runnable runnable):
  userConnection(runnable) {
        UserPool.UserThread freeUser
        = pool.getUserThread();
        if (freeUser == null){
              freeUser = new UserPool.UserThread();
        }
        freeUser.setUser(runnable);
        activeUser++;
        if(activeUser > population)
           System.out.println("The active user:"+activeUser
           +" exceeds the maximum user:"+population);
        return freeUser;
   }
  pointcut communicate(UserPool.UserThread user)
  : execution(void UserPool.UserThread.run()) &&
  this(user);
  void around(UserPool.UserThread user)
  : communicate(user) {
      while(true) {
         proceed(user);
         pool.putUserThread(user);
         activeUser--;
         synchronized(user) {
               try {
                  user.wait();
               } catch(InterruptedException ex) {}
         }}}
}
```

Figure 9 – A Population Aspect : PopulationAJ

The APEF can measure the population by using a population aspect, PopulationAJ depicted in Figure 9. A variable activeUser has current active users on the server. PopulationAJ checks if the value of activeUser is more than 1,000. Note the pointcut userConnection(Runnable runnable) and around advice Thread around(Runnable runnable).userConnection picks out join points at when Thread.new(Runnable) is called. When those join points are reached around(Runnable runnable) is executed. If an available thread is in a thread pool, it is assigned for a user. Otherwise, a new thread is created. The number of current active users increases by one because one thread has been activated. The web server is notified if the number of active users exceeds the population. A second pointcut is communicate(UserPool.UserThread user). A selected join point is execution(void UserPool.UserThread.run()). Whenever run() of class UserThread is executed, around advice void around(UserPool.UserThread user) is executed. A thread is put into the pool and the number of active users decreases by one if the session between user and server is disconnected. The thread in the pool remains in a wait state.

5.3. An Aspect for Throughput

Throughput is considered in a third example. According to OMG [4], throughput is defined as the rate at which the resource performs its function. In this example, we define throughput as the average number of frames transferred per second from a video server to a video player. In Figure 6, VideoServer calls a method sendFrame() of VideoPlayer to send frames of a movie. To measure the throughput of frame-sending, it is necessary to know how many frames are transferred per second. ThroughputAJ in Figure 10 uses before and after advice to measure execution time of sendFrame() of VideoServer. The number of transferred frames is given by a variable, numOfFrames. The measured execution time needs to be converted into seconds. As the number of frames is divided by the execution time, the throughput can be calculated. A variable throughput in after advice stores the throughput of frame-sending.

```
public aspect ThroughputAJ{
   long begin,end,rTime;
   int throughput=0;

    pointcut sendFrameMethod(VideoServer vs)
   :execution(void VideoServer.sendFrame())
    && target(vs);
     before(VideoServer vs) : sendFrameMethod(vs) {
          begin=System.currentTimeMillis();
   }
  after(VideoServer vs) returning : sendFrameMethod(vs) {
         end=System.currentTimeMillis();
         rTime=end-begin;
         int sec=(int) (rTime/1000);
        throughput = vs.numOfFrames()/sec;
        System.out.println("The throughput is "+ throughput
                                +" frames/seconds");
   }
}
public class VideoServer{
     VideoPlayer player=null;
     int numOfFrames=1000;
     public VideoServer(VideoPlayer vplayer){
          player=vplayer;
     }
     public void sendFrame(){
        for(int i=0;i<numOfFrames;i++){
             player.sendFrame(new VideoFrame());
         }
      }
     public int numOfFrames(){
        return numOfFrames;
     }
  }
public class VideoPlayer{
     public void sendFrame(VideoFrame frame){
       //do something
     }
}
```

Figure 10 – A throughput aspect : ThroughputAJ

6. Conclusions and Future Work

The main goal of APEF is to generate performance aspects which can be used to evaluate performance requirements. In this effort, we assume the performance requirements are specified in UML models using the performance profiles. The UML models consist of primary (analysis and design) and performance models. The primary model represents the structural and behavioral perspectives of software systems and usually focuses on modeling functional requirements of software systems. The performance model specifies performance information using the performance profiles. Both models are the input of APEF. APEF applies AOP principles into software performance evaluation. Performance as a cross-cutting concern can be separated from the primary models and implemented as an aspect. Generated performance aspects in AspectJ are based on the performance model. The results of the performance evaluation can be reflected to the performance model. The entire procedures could be iterated until performance is acceptable.

This is on-going research and the APEF is under development. We continue to implement the performance AspectJ code templates which are intended to fully support the performance analysis domain model. We anticipate considerable effort in providing a seamless performance evaluation framework from design to implementation phase.

While APEF does not provide a performance prediction framework, it does provide a credible performance evaluation framework. For the performance prediction, a mathematical or formal performance model will be necessary. Future research is necessary to build on APEF and enhance its functionality to include module for the performance prediction. We also plan to extend APEF to support other software quality attributes such as security, concurrency, and dependency.

References

[1] Unified Modeling Language (UML) Tutorial, http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm

[2] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, Non-Functional Requirements in Software Engineering, Kluwer Academic Publishing, October 1999.

[3] G. P. Gu and D. C. Petriu, "Early Evaluation of Software Performance based on the UML Performance Profile," Proc. 2003 Conf. of the Advanced Studies Conf. on Collaborative research, pp. 66 – 79, October 2003.

[4] Object Management Group (OMG), UML profile for Schedulability, Performance, and Time Specification, January 2005.

[5] A. Bennett and A. Field, "Performance Engineering with the UML Profile for Schedulability, Performance and Time:a Case Study," 12th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.

[6] G. Kiczales J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming," Proc. of the 11th European Conf. on Object-Oriented Programming, pp. 220-242, June 1997.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," Proc. of the 15th European Conf. on Object-Oriented Programming, pp.327-353, June 2001.

[8] AspectJ, http://www.eclipse.org/aspectj

[9] R. Miles, AspectJ Cookbook, O'REILLY, 2005.

[10] M. M. Kandé, J. Kienzle, and A. Strohmeier, "From AOP to UML-A Bottom-Up Approach," AOSD'2002 Workshop on Aspect-Oriented Modeling with UML, April 2002.

[11] S. Balsamo, A. D. Marco, P. Inveradi, and M. Simeoni, "Model-Based Performance Prediction in Software Development: A Survey," IEEE Transactions on Software Engineering, vol. 30, no. 5, May 2004.

[12] L. B. Arief and N. A. Speirs, "A UML Tool for an Automatic Generation of Simulation Programs, " Proc. 2nd Int'l Workshop on Software and Performance, pp. 71 – 76, September 2000.

[13] D. Park and S. Kang, "Design Phase Analysis of Software Performance Using Aspect-Oriented Programming," Proc. 5th Aspect-oriented Modeling Workshop in conjunction with U\ML 2004, 2004.

[14] O. Aldawud, T. Elrad, and A. Bader, "UML Profile for Aspect-Oriented Software Development," Proc. of 3rd International Workshop on Asepct-Oriented Modeling, March 2003.

[15] M. Basch and A. Sanchez, "Incorporating Aspects into the UML," Proc. of Third International Workshop on Aspect-Oriented Modeling, March 2003.

[16] D. Stein, S. Hanenberg, and R. Unland, "On Representing Join Points in the UML," 2nd International Workshop on Aspect-Oriented Modeling with UML, UML 2002, September 30, 2002

[17] W. Ho, F. Pennaneac'h, and N. Plouzeau, "UMLAUT: A Framework for Weaving UML-based Aspect-Oriented Designs," TOOLS '00: Proc. of the Technology of Object-Oriented Languages and Systems (TOOLS 33), pp. 324-334, IEEE Computer Society, 2000.

[18] R. Bodkin, AOP@Work: Performance monitoring with AspectJ, http://www-128.ibm.com/developerworks/java/library/j-aopwork10/

[19] D. J. Pearce, M. Webster, R. Berry, and P. H.J. Kelly, "Profiling with AspectJ," Software: Practice and Experience, to appear, 2006.

[20] Object Management Group (OMG), XML Metadata Interchange (XMI) specification, version 1.2.