# Implementing Self-Adaptive Software Architecture by Reflective Component Model and Dynamic AOP: A Case Study

Yuankai Wu, Yijian Wu, Xin Peng, Wenyun Zhao
School of Computer Science and Technology
Fudan University
Shanghai, China
{072021143, wuyijian, pengxin, wyzhao}@fudan.edu.cn

*Abstract*—**Architecture-based method, which implements self-managing characteristics by dynamically configuring or reconfiguring the runtime architecture, has been widely accepted as a promising approach for self-adaptive systems. Some reflective architecture and component models like Fractal are proposed to support dynamic architecture adaptations through introspection and reconfiguration APIs. We believe dynamic AOP (Aspect-Oriented Programming) should also be employed as a complementary means for crosscutting adaptations. In this paper, we conduct a case study on implementing self-adaptive software architecture by reflective component model (Fractal) and dynamic AOP in an industrial Web-based system. With the case study, we hope to evaluate pros and cons of reflective component and dynamic AOP in implementing self-adaptive software architecture. In our case study, we identify four typical self-adaptation scenarios with the solutions. We also evaluate both approaches in terms of effectiveness, runtime efficiency and development/maintenance efforts. Our case study shows that reflective component model and dynamic AOP are effective for structural architecture adaptations, but have shortages in flexibility and do not support behavioral adaptation.**

***Keywords-****self-adaptive architecture; reflective component; Fractal; dynamic AOP*

## I. INTRODUCTION

Traditional software is implemented under static decisions in analysis and design time based on assumptions about the requirements and runtime environment. Therefore, any unanticipated changes to the requirements or runtime environment will lead to a manual maintenance process, which is unacceptable in critical systems. Self-adaptive systems promise to provide the capability of runtime adaptations to various changes. They can configure and reconfigure themselves at run time to handle such things as changing user needs, system intrusions or faults, a changing operational environment, and resource variability [4]. This capability of self-adaptation is achieved by a series of self-managing characteristics, including self-configuring, self-healing, self-optimizing, self-protecting [5][6].

Although there are also language-level or network-level methods for self-adaptation, architecture-based methods have been widely accepted as more promising approaches, since they provide the required level of abstraction and generality to deal with the challenges posed [7]. Architecture-based self-adaptive software implements the self-managing characteristics by dynamically configuring or reconfiguring the runtime architecture, e.g. adding/removing/replacing components or modifying component connections. In the three-layer reference model for self-adaptive architecture proposed by Kramer and Magee [7], three architecture layers are identified: goal management, change management, component control. The bottom layer, component control, consists of a set of interconnected components and provides facilities for status report and architecture adaptation. In this paper, we focus on the implementation techniques in the component control layer.

In traditional development paradigms like object-oriented methods, there do not exist explicit runtime architectural elements like components and connectors to enable runtime adaptations. Therefore, some reflective architecture and component models like Fractal [1][2] are proposed to support dynamic architecture adaptations through introspection and reconfiguration API. Fractal supports hierarchical architecture definition and interface-based composition mechanism with dynamic reconfiguration. However, just as Aspect-Oriented Programming (AOP) [3] complements Object-Oriented Programming (OOP) by providing modularization of crosscutting concerns [8], reflective component model should also be complemented by some kind of modularized adaptation mechanism for crosscutting architectural elements. Dynamic AOP, which provides runtime dynamic configuration and reconfiguration on aspect weaving, can support this kind of crosscutting adaptations.

This paper presents a case study on an industrial Web-based system, which implements a self-adaptive software architecture with a reflective component model (Fractal) and dynamic AOP. The system is derived from an existing Public Service system for Self-taught Examination (PSSE), which provides on-line services, such as examination registration, grade query and information publication, to self-taught students. We are trying to evaluate strengths and weaknesses of reflective component and dynamic AOP in implementing self-adaptive software architecture. Four typical self-adaptation scenarios are identified, covering both self-healing from environmental failures and self-optimizing for an unanticipated environment. Local adaptations and crosscutting adaptations are also concerned in implementation. For each scenario, we present a solution and make an evaluation for effectiveness, runtime efficiency and development/maintenance effort.

In Section II, we introduce some background of our case study. Some related work is introduced in Section III. Then, we describe the four self-adaptation scenarios and their solutions in Section IV, followed by evaluations in Section V. We conclude the paper with some planned future work in Section VI.

## II. BACKGROUND

### A. An Overview of PSSE

PSSE is a Web-based public service system with broad users and typical high-assurance requirements. PSSE provides services for both self-taught students and administration officers. As a real web-based industrial business application, it involves typical self-adaptive requirements that are representative of common cases. Typical self-adaptation cases will be described in detail in Section IV.

PSSE is a J2EE application developed in Java/JSP. The original PSSE implementation has a typical object-oriented architecture consisting of 7 java packages, 65 java source files, 108 JSP web pages and several configuration files.

The original implementation cannot support runtime self-adaptations, since most interactions between objects are hard-coded. In order to conduct the case study, we first refactor the implementation with a component-based design, where original hard-coded object interactions are re-written into interface-based component compositions. Part of the component-based PSSE architecture after refactoring is shown in Figure 1, using the hierarchical Fractal[1][2] style.

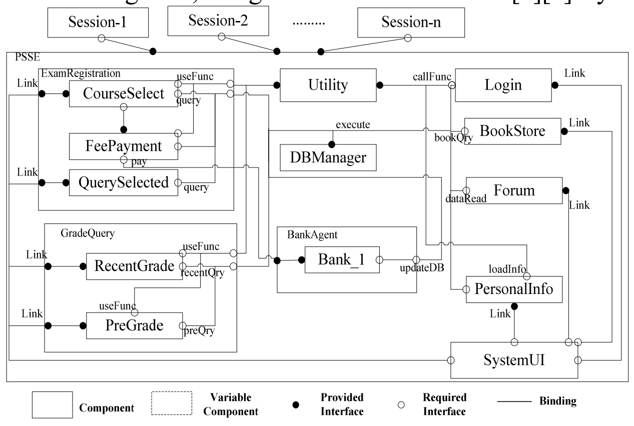

Figure 1. PSSE Architecture Model after Refactoring (partial)

In the architecture diagram, components are represented by nested blocks with solid and hollow circles denoting provided and required interfaces respectively. The whole system PSSE can be regarded as a root component. Connections between interfaces represent compositions between two interacting components or delegations between composite components and their sub-components. Among the components, ***SystemUI*** is the user interface component comprised of JSP web pages; ***Utility*** provides utility functions; ***DBManager*** is the database access component; others are business components for various business services like grade querying, information consultation and online payment.

### B. Reflective component model: Fractal

Fractal is a component model developed by France Telecom R&D and INRIA, and distributed through the ObjectWeb consortium. We choose Fractal over other component models because it is reflective, and is highly dynamic. Fractal component model relies on some classical concepts: components as run-time entities, interfaces as the only interaction points between components in terms of required/client and provided/server interfaces, bindings as communication channels between component interfaces [2]. Figure 2 shows a Fractal component model. In the model, Component = Membrane + Content, Content = set of (sub-)components, Membrane = composition and reflection behavior. The reflection in Fractal is based on the composite component mechanism, which has an internal architecture consisting of a series of subcomponents and their interconnections. The internal architecture of a composite component can then be manipulated at runtime to implement the architecture-level reflection.

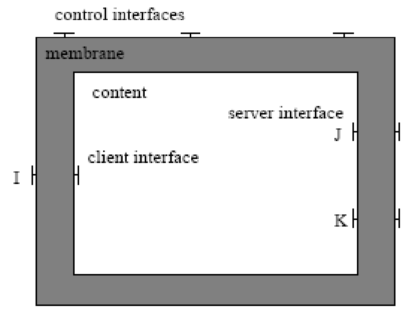

Figure 2. Fractal Component Model [1]

In our case study, we use Julia, a java-based implementation framework for Fractal, as the implementation of the component model. More information about Fractal and Julia can be found from the Fractal website [9].

### C. Dynamic AOP

Aspects enable modularization of concerns such as logs that crosscut multiple classes and objects. Dynamic AOP extends static AOP by providing dynamic weaving mechanism. It can weave aspects at runtime without stopping the whole system.

Spring AOP [3] is a dynamic AOP framework implemented in pure Java. It is proxy based. It can use either standard J2SE dynamic proxies or CGLIB [10] for AOP proxy. Also, Spring AOP is schema based. Aspects and pointcut definitions are centralized in an XML file. The XML file can be dynamically modified, thus the aspects and pointcuts can be dynamically changed. These mechanisms make Spring AOP flexible and easy to maintain. Spring AOP can be used independently without Spring framework, it has friendly development tool for new users. So it is easy to learn and apply in a real project.

## III. Related Work

Before our case study, there have been some works (e.g. [11][12]) reported focusing on implementation techniques for self-adaptive systems. The approach reported in [11] gives us an interesting framework based on Fractal platform to separate the adaptation concern from business code. B. Morin et al. [13] present an approach for managing the complexity of dynamically adaptive systems. The approach combines aspect-oriented and model-driven techniques to enable adaptation validation at runtime. Sicard et al. [14] report a self-repair case and propose that components provide a way of building architecture based management systems which provide a way of adapting it to changing environments.

These studies, however, usually focused on how to develop an adaptive system with certain techniques. Our study has a different focus on evaluating related techniques in a real industrial product.

## IV. Self-adaptation implementations for fault-tolerance

In the case study, we identify and implement four self-adaptation scenarios as listed in Table 1. In the first scenario, dynamic tradeoff is conducted to improve response time with the cost of lowered security level. In the second scenario, the load keeps growing making nonfunctional optimization insufficient to ensure an acceptable response time, then functional optimization is conducted to unload some secondary functions to ensure the quality of kernel services like course registration. The third scenario can be understood as self-healing from environmental failures, and the last one as self-optimizing by dynamic partner service replacement. The first two scenarios are internal optimization for high concurrent load, while the last two are self-healing adaptations to environmental failure or quality degradation.

### A. *High-load concurrent accesses*

Response time and throughput of the system may degrade greatly when concurrent accesses increases. This degradation can be dealt with by loosening some nonfunctional requirements (e.g. security). In PSSE, logging is employed for security by recording operations for each service request. When the server is overloaded, security level can be lowered for more critical quality properties for kernel services.

In this scenario, logging component is loaded and unloaded dynamically according to system load. This is implemented as an aspect using dynamic AOP. Before adaptation, the logging component crosscuts a lot of business components (Figure 3). Figure 4 shows an implementation in Spring AOP with a modified pointcut file (left) and code that changes some weaved connections around the logging component.

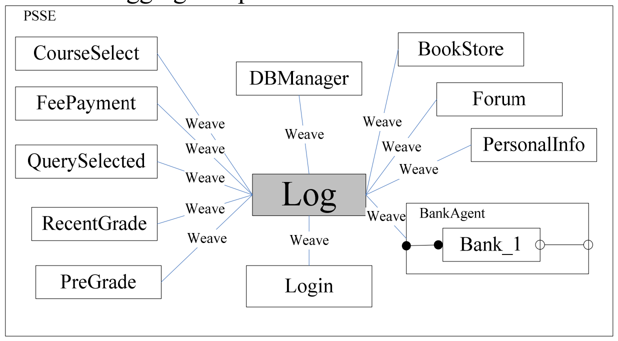

Figure 3. Logging component before adaptation

TABLE I. The four self-adaptation scenarios used in our study

| Scn. | Scenario Description | Type | Adaptation Policy |
|---|---|---|---|
| 1 | high-load concurrent accesses: the number of concurrent accessing users grows beyond the anticipated peak value 150 | Nonfunctional Optimization | lower the security level by unloading system logging to improve the response time |
| 2 | high-load concurrent accesses (worse): the number of concurrent accessing users further grows over 200 | Functional Optimization | unload some secondary services to ensure the quality of critical services |
| 3 | database not available: the required database service is not available | Environment Failure | switch to an alternative solution that does not depending on the failed database |
| 4 | low-quality payment service: quality of the default electronic payment service is low | Environment Optimization | switch to an alternative payment service provider |

```
<bean id="logPointcutAdvisor"
class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
        <property name="advice">
                <ref local="MyInterceptor" />
        </property>
        <property name="mappedNames">
                <list>
                        <value>Login</value>
                        <value>DBManager</value>
                </list>
        </property>
</bean>
```

```
public void uninstallLogger() {
        String F = AdaptiveParameter.bean;
        modifyBeanNoLog(F); //modify bean.xml
        reloadBean(F);  //reload new bean
}

public void installLogger(){
        String F = AdaptiveParameter.bean;
        modifyBeanLog(F);
        reloadBean(F);
}
```

Figure 4. Pointcut file and dynamic weaving example in Spring AOP

## *B. High-load concurrent accesses (worse)*

In Scenario 1, security level is lowered to alleviate the problem of throughput and response time degradation when the server is overloaded. When the concurrent accesses keep growing, the nonfunctional tradeoff adopted in Scenario 1 is insufficient, so functional tradeoff has to be considered. In PSSE, examination registration, by which a student can register himself and selected subjects to the current examination, is the most critical service. Especially when the deadline of the current examination is approaching, it is unacceptable that many students cannot register the examination in time due to the overloaded server. Therefore, in order to ensure the quality of critical services, functional tradeoff can be adopted to unload some secondary services like forum and grade query. Thus, limited server resources can be utilized prior to some critical services.

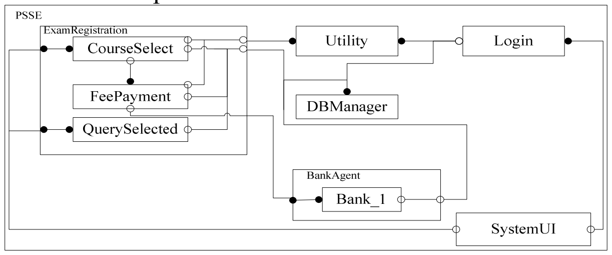

Figure 5. PSSE architecture after unloading some secondary services

```
public class Init{
     public static Component psse;
     public static Component forumComp; ......
     public static void archInit() throws Exception {
     Component boot = Fractal.getBootstrapComponent();
     defineType(boot);
     Fractal.getContentController(psse).addFcSubComponent(forumComp);
     Fractal.getBindingController(psse).bindFc("forum",
               forumComp.getFcInterface("forum"));
}
```

```
public void removeForum() throws Exception{
      Fractal.getLifeCycleController(psse).stopFc();
      Fractal.getBindingController(psse).unbindFc("forum");
      Fractal.getContentController(psse).removeFcSubComponent(forumCo
            mp);
      Fractal.getLifeCycleController(psse).startFc();
}
```

Figure 6. Implementation of component binding and removing by Fractal

Figure 5 shows the PSSE architecture after unloading some secondary services. Figure 6 shows the implementation code of component binding (the upper part) and removing (the lower part) by Fractal. It can be seen that in component removing, the component should first be unbound, and then removed.

## *C. Database not available*

PSSE is a typical database-based information system. In the original implementation, most of the services will collapse on database unavailability. This failure is unacceptable during examination registration service, especially when the registration deadline is approaching. In this scenario, the system will switch to another persistency solution (e.g. local files) to replace a database temporarily. Figure 7 shows a runtime architecture, where component ***LocalFileRecord*** takes the place of the original ***DBManager*** (not shown).

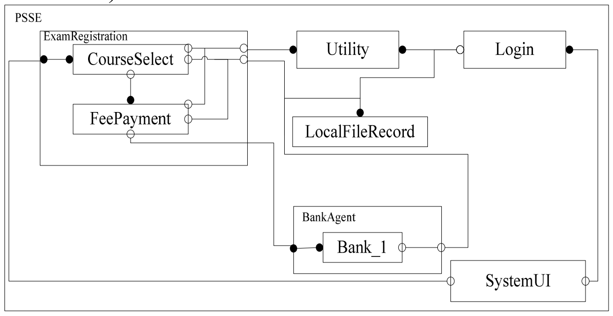

Figure 7. SSE architecture after switching to the solution without DB

The implementation is shown in Figure 8, in which the left part is the brief implementation of ***LocalFileRecord*** and the right part is the adaptation code. The local-file-based solution uses local files to record SQL statements instead of executing them. As most of the business data are fetched to the memory when the system starts up, critical services may continue running without accessing database. After the database service recovers, besides turning the persistence component back to ***DBManager***, an additional offline data transfer process will be conducted to make the SQL statements recorded in local files persistent in the database. The recorded username/password will be checked and if wrong related SQL statements will be ignored. Moreover, this alternative solution can not support some other services like grade query, since there is not local copy of grade data. Therefore, in this scenario, the alternative solution provides services at a degraded quality level.

```
Public class localFileRecordDB implements Intf_DB{
      public void executeUpdate(String sql) throws Exception{
         BufferedWriter    output    =    new    BufferedWriter(new
             FileWriter(AdaptiveParameter.file, true));
         output.write(sql);
         output.newLine();
         output.flush();
         output.close();
      }
}
```

```
public void switchDB() throws Exception{
     Fractal.getLifeCycleController(Init.dbComp).stopFc();
     Fractal.getBindingController(Init.dbComp).unbindFc("db");
     Fractal.getContentController(Init.dbComp).removeFcSubComponent(Init.dbmanagerComp);
     Fractal.getContentController(Init.dbComp).addFcSubComponent(Adapt.localFileRecordComp);
     Fractal.getBindingController(Init.dbComp).bindFc("db",
          Adapt.localFileRecordComp.getFcInterface("db"));
     Fractal.getLifeCycleController(Init.dbComp).startFc();
}
```

Figure 8. Implementation of component replacement by Fractal

### D. *Low-quality payment service*

In PSSE, successful online payment is a prerequisite for examination registration, so electronic payment service provided by related banks is a critical external service resource. In real application environment, the runtime quality of the payment service, which greatly influences the quality of examination registration service, may vary a lot. Therefore, if the quality of the payment service degrades to certain level, an adaptation policy will be enforced to replace the default payment service with another service provided by a different bank. In PSSE, this adaptation means to replace the local bank agent component with another one for the new service provider as shown in Figure 9. Figure 10 shows an implementation with Fractal.

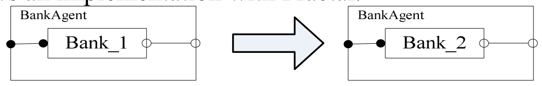

Figure 9. Architectural adaptation when Bank_1 is not available

```
Public void switchBank() throws Exception {
      Fractal.getLifeCycleController(Init.bankAgentComp).stopFc();
      Fractal.getBindingController(Init.bankAgentComp).unbindFc("bank");
      Fractal.getContentController(Init.bankAgentComp).removeFcSubComp
onent(Init.bank_1_Comp);
      Fractal.getContentController(Init.bankAgentComp).addFcSubCompone
nt(Adapt.bank_2_Comp);
      Fractal.getBindingController(Init.bankAgentComp).bindFc("bank",Ada
pt.bank_2_Comp.getFcInterface("bank"));
      Fractal.getBindingController(Adapt.bank_2_Comp).bindFc("db",
             Fractal.getContentController(Init.bankAgentComp).getFcInternal
             Interface("db")); // subComponent's interface proxy
      Fractal.getLifeCycleController(Init.bankcComp).startFc();
}
```

Figure 10. Implementation of component replacement by Fractal

### E. *Summary*

In the given scenarios, Fractal provides self-adaptation supports by its APIs for component lifecycle management and binding configuration, while dynamic AOP supports crosscutting adaptations by runtime aspect weaving configurations. Implementation techniques and their impact on the runtime architecture are listed in Table II. As a component consists of several Java classes, affected classes may be much more than affected components.

TABLE II. SUMMARY OF IMPLEMENTATIONS IN THE FOUR SCENARIOS

| Scn. | Adaptation Implementation | #Component Affected | #JAVA class Affected | #Binding Altered | #Components Removed |
|---|---|---|---|---|---|
| 1 | Dynamic AOP: (un)weave aspects | 16 | 31 | 11 | 1 |
| 2 | Fractal: component removing | 9 | 33 | 19 | 6 |
| 3 | Fractal: component replacement | 11 | 38 | 23 | 7 |
| 4 | Fractal: component replacement | 3 | 4 | 4 | 1 |

## V. EVALUATION AND DISCUSSION

Based on our case study, we evaluate reflective component and dynamic AOP in implementing self-adaptive software from the following three aspects.

### A. *Effectiveness*

Traditional adaptive software is implemented by hard-coded adaptation logic like "if-else" statements, tangled with business code, very difficult to revise, reuse and maintain [4]. Self-adaptive architecture provides an architecture-based way for runtime adaptations. In general, Fractal supports most of the component-based runtime adaptations such as adding/removing/replacing components and binding connection reconfiguration. However, all these are structure-based adaptations and behavioral adaptation is not supported by Fractal. Behavioral adaptation means adapting the interaction protocols, e.g. interaction sequences and styles, among components and within components even without any structure adaptations. This kind of behavioral adaptation requires that the component control element, e.g. the Membrane in Fractal, can further control the interaction protocol and process, just as the component model proposed in our previous work [15]. Moreover, the architecture reconfiguration in Fractal is strictly interface-based, which means composition can only be conducted between two components with exactly the same syntax interface. This limits the flexibility of architecture adaptation.

### B. *Runtime efficiency*

In order to evaluate the runtime efficiency, we conduct an experiment to compare performance of the original and the adaptive PSSE systems. The experiment environment is Tomcat 5.0 running on a test server with Pentium D 2.80GHz CPU and 1G memory. JMeter[16], a stress testing tool, is used to simulate continuous concurrent accesses of 25~500 users to the PSSE system in 50 seconds. In each access thread, a series of usual operations like logon, access forum, query previous grade, query recent grade, query selected courses, exam registration, log out are executed repeatedly and the response time of each operation is recorded.

Figure 11 shows a comparison in average response time between the original and the self-adaptive implementations. Initially, when concurrent accesses are lower than 150, performance of the self-adaptive PSSE version is a little worse due to the additional cost of the adaptation implementation. When the concurrent accesses exceed 200, the first and second adaptation scenarios are triggered respectively, which leads to a better performance in the self-adaptive version. To an extreme, the self-adaptive version has a performance improvement of 36.12% (2985 vs 4673 milliseconds) when 500 concurrent users are online.

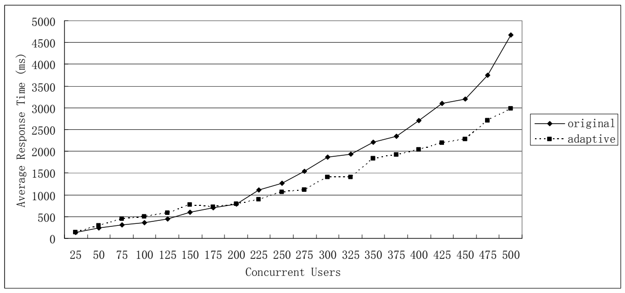

Figure 11. Comparison of average response time for examination registration

### C. *Development and maintenance effort*

Development and maintenance effort is also an important issue in real world software development. According to our experience, developing a product with Fractal and dynamic AOP requires some but not so much extra learning effort.

The application of Fractal and dynamic AOP requires an explicit and well-designed component-based architecture like the architecture shown in Figure 1. This kind of architecture designs involves design principles like explicitly-defined interface, interface-based composition, and hierarchical design. The application of dynamic AOP further improves the status for the principle of SOC (separation of concerns). Thanks to these good design principles, the maintenance effort can be reduced greatly, even for non-adaptive systems. Both Fractal and dynamic AOP can support the separation of business logics and self-adaptation implementations, which helps future maintenance greatly.

## VI. Conclusion and Future Work

In order to evaluate existing techniques for self-adaptive system, we conduct a case study of implementing self-adaptive software architecture by reflective component model and dynamic AOP in an industrial Web-based system. In the case study, we identify four typical self-adaptation scenarios with corresponding self-adaptation solutions. Then, we evaluate the strengths and weaknesses from three aspects of effectiveness, runtime efficiency and development/maintenance effort.

Our case study shows that reflective component model and dynamic AOP are effective for implementing self-adaptive systems. The overall runtime quality of the self-adaptive system can be greatly improved with acceptable cost. Moreover, involved good design principles, such as SOC, interface-based and hierarchical design, make software maintenance easier, although some additional learning and modification efforts are needed. Our study also shows that reflective component model and dynamic AOP do not support behavioral adaptations, which limits the feasibility and flexibility of these mechanisms.

Our case study focuses on implementation techniques of runtime adaptations. Other issues like monitoring, analysis and planning in self-adaptive systems are not specially addressed. In our future work, we will further conduct more complete case study of self-adaptive systems covering all the MAPE (Monitoring, Analysis, Plan, and Execution) loops.

## Acknowledgment

This work is supported by National Natural Science Foundation of China (Grant No. 90818009), National Program on Key Basic Research Project (Grant No. 2006CB303002), and Shanghai Leading Academic Discipline Project (Grant No. B114).

## References

[1] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, J.B. Stefani. The Fractal Component Model and Its Support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. Software Practice and Experience, 2006, 36: pp 1257-1284.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, J.B. Stefani. An Open Compoent Model and Its Support in Java. CBSE2004, LNCS 3054, pp.7-22.

[3] SpringAOP, http://www.springsource.org/.

[4] Betty H.C. Cheng, Rogério de Lemos, Stephen Fickas, D. Garlan, M. Litoiu, J. Mage, et al. SEAMS 2007: Software Engineering for Adaptive and Self-Managing Systems. International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'07), June 2007, p 1-1.

[5] IBM Corporation. An architectural blueprint for autonomic computing. http://www.ibm.com/developerworks/autonomic/library/ac-summary/ac-blue.html.

[6] A. Lapouchnian, Y. Yu, S. Liaskos and J. Mylopoulos. Requirements-Driven Design of Autonomic Application Software. In Proceedings of CASCON'06. Toronto, 2006, pp 7-21

[7] Jeff Kramer and Jeff Magee. Self-Managed Systems: an Architectural Challenge. Future of Software Engineering (FOSE'07), 2007, pp 259-268

[8] G. Kiczales, et al. Aspect-Oriented Programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997), 1997, pp 220-242

[9] Fractal Home, http://fractal.objectweb.org/

[10] CGLIB: http://cglib.sourceforge.net/

[11] P. David and T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In SC'06: 5th Int. Symposium on Software Composition, LNCS 4089, Vienna, Austria, 2006, pp 82–97.

[12] R. Wolfinger, S. Reiter, D. Dhungana, P.Grunbacher, and H. Prahofer. Supporting runtime system adaptation through product line engineering and plug-in techniques. In ICCBSS'08: 7th Int. Conf. on Composition Based Software Systems, 2008, pp 21-30.

[13] B. Morin, O. Barais, G. Nain, J .M. Jezequel. Taming Dynamically Adaptive Systems Using Models and Aspects. In ICSE 2009, the 31th international conference on Software engineering, 2009, pp 122-132.

[14] S. Sicard, F. Boyer, N. De. Palma. Using Components for Architecture-Based Management: The Self-Repair case. In ICSE 2008: the 30th international conference on Software engineering, 2008, pp 101-110.

[15] Xin Peng, Yijian Wu, Wenyun Zhao. A Feature-Oriented Adaptive Component Model for Dynamic Evolution. The 11th European Conference on Software Maintenance and Reengineering (CSMR'07).

[16] JMeter, http://jakarta.apache.org/jmeter/