Efficient and Transparent Instrumentation of Application Components Using an Aspect-Oriented Approach

Markus Debusmann[1] and Kurt Geihs[2]

[1] Fachhochschule Wiesbaden, University of Applied Sciences
Department of Computer Science, Distributed Systems Lab
Kurt-Schumacher-Ring 18, 65197 Wiesbaden, Germany
debusmann@informatik.fh-wiesbaden.de
[2] Berlin University of Technology
Intelligent Networks and Management of Distributed Systems
Einsteinufer 17, 10587 Berlin, Germany
geihs@ivs.tu-berlin.de

Abstract. The increasing significance of Service Level Management (SLM) strongly requires an appropriate instrumentation of application components in order to monitor compliance with the defined Service Level Objectives (SLOs). The manual instrumentation of application components is very costly and error-prone and thus rather inefficient. This paper presents an approach for using aspect-oriented programming techniques for efficiently and transparently instrumenting application components. The approach is applied to the interference sensitive area of performance monitoring using the Application Response Measurement (ARM) API. Experiments with a prototype have revealed that our aspect-oriented approach fits well to the integration of instrumentation code into application components and that the runtime overhead is only slightly higher than the overhead of a manual instrumentation.

1 Motivation and Related Work

Over the past years, economic pressure has forced enterprises to outsource many IT services and purchase them from external service providers. Quality-of-Service (QoS) parameters agreed on by service providers and their customers are laid down in a contract, called service level agreement (SLA) [1, 2, 3]. Typical QoS parameters specified in SLAs define availability criteria and performance-related metrics, e.g. response times. The fulfilment of such an SLA has to be monitored at run-time by both the customer and the service provider. Customers are primarily interested in short end-user response times and high service availability. Providers are interested in a more fine-grained view of the interrelated performance metrics of the components constituting their services, especially when using the same service infrastructure for different customers at the same time.

For computing high-level SLA parameters, such as service response times and availability, the application components within the service provider domain

M. Brunner and A. Keller (Eds.): DSOM 2003, LNCS 2867, pp. 209–220, 2003.

have to be instrumented appropriately for management purposes. Instrumenting applications incurs additional cost since instrumentation is a non functional requirement that is not part of the business logic interface. Often, instrumentation is added on top of existing applications in an ad-hoc manner as described in [4]. In [5] an integrated software development process for applications and their management infrastructure is described. The development of the management functionality runs in parallel to the development of the normal software. For implementing management solutions based on CIM [6], developers are additionally supported by a design patterns catalogue that contains reusable patterns for CIM models.

Alternative approaches aim for realising instrumentation in a transparent manner, e.g., by instrumenting middleware components and platforms. [7] presents an approach for modifying an EJB server in a way that EJBs are instrumented automatically while they are deployed. Response times of method invocations are automatically monitored using the Application Response Measurement (ARM) API defined by the OpenGroup [8]. In addition, EJBs are instrumented via Java Management Extensions (JMX) [9] to expose configuration information. In [10] CORBA applications are instrumented based on CORBA Portable Interceptors which allow to insert any kind of instrumentation code that is executed at certain points of method invocations. The insertion of the Portable Interceptor code into the application causes only minimal effort for the application developer. He only has to add two additional lines of code to the source code of the component which is to be instrumented. [11] presents an approach for an automated instrumentation of component-based applications. The approach is limited to measuring and correlating response times similar to the Application Response Measurement (ARM) API defined by the OpenGroup.

The examples described above show that there is a certain tradeoff between the degree to which the instrumentation can be automated and the level of detail obtained by the instrumentation. The more generic the instrumentation approach the more abstract is the achievable data. On the other hand, the more specific information shall be extracted the higher the instrumentation costs for the application developer. Obviously, if fine-grained monitoring is the goal, then the instrumentation needs to be woven into the code of the application components. This is where aspect-oriented programming (AOP) can help.

This paper presents an aspect-oriented instrumentation approach for application components. Aspect-orientation helps to solve the shortcomings of the existing instrumentation approaches by providing transparency for the application developer and furthermore offering the opportunity to monitor management data at any level of detail. In addition, it is not limited to monitoring a single environment like CORBA or EJB.

The paper is structured as follows: Section 2 gives a brief introduction to the concepts of aspect-oriented programming and introduces AspectJ as an example of an aspect-oriented development environment. In Section 3 we present how the aspect-oriented paradigm can be applied to simplify the monitoring of distributed applications. Section 4 discusses performance measurements that under-

line the efficiency of the aspect-oriented instrumentation compared to a manual instrumentation. Section 5 concludes the paper and gives an outlook to future work.

2 Aspect-Oriented Programming (AOP) and AspectJ

The design of large applications is arduous since decisions made in early modelling stages influence later phases, i.e. implementation and maintenance. Hence much effort in the past has been made to employ proper software analysis and design methods. Typically these methods are tied to a distinct programming paradigm, e.g. imperative or object-oriented, and they attempt to structure the model accordingly. The aspect-oriented programming paradigm (AOP) [12] claims that although conventional analysis and design methods try to partition a given problem into self-contained, encapsulated entities, this kind of partitioning is not always feasible. Dependencies between modelled entities break the desired encapsulation and thus make the design hard to deploy and even harder to reuse. AOP aims at a conceptual understanding of the "cross-cutting" of responsibilities through separate entities. For that purpose, the AOP distinguishes aspects and components.

A component is an entity which encapsulates a distinct responsibility. Components can be combined with other components to achieve a distinct behaviour. They interact through well defined interfaces.

In contrast to components, aspects are not encapsulated. They cross-cut each other, preventing clear encapsulation. Aspects typically reflect non-functional issues of a system, e.g., error handling, performance tuning, or synchronisation.

The proposed solution to the coexistence of aspects and components in a system is the introduction of aspect languages and a corresponding aspect weaver. Aspect languages are specialised languages, suitable for modelling and expressing the distinct properties of an aspect and its connection with other aspects. The aspect weaver is a tool which takes all aspect specifications of a system and generates a corresponding program.

So far, most aspect-oriented approaches dealt with distribution and synchronisation aspects in distributed systems. Recently, the integration of QoS management in distributed applications has become a target for AOP-based systems [13, 14].

AspectJ [15], originally developed at Xerox Parc, is an implementation of aspect-oriented programming paradigm for the Java language. In the following, a short overview of the AspectJ's basic concepts and constructs is given; details are provided in [16]. AspectJ defines the concept of a *join point* which represents a well-defined point within the program flow, e.g., a method call, a constructor call, referencing of a field, etc. *Pointcut designators*, or simply *pointcuts*, are used to identify certain join points within the program flow. Pointcuts can also be combined using filters to define more complex expressions. For example the pointcut

```
pointcut foo() : call (void ClassA.methodA (int)) ||
                 call (* ClassB.get* (..));
```

identifies every invocation of `methodA` in `ClassA` as well as every invocation of `ClassB`'s methods that start with `get`, regardless of its return type and its parameter list.

Advices define the additional code, typically implementing the cross-cutting concern, that should be executed when a join point is reached. Pointcuts are used within the definition of an advice to identify the join point. Three different advice types are distinguished: *before advice* run when their joint point is reached, *after advice* run after their joint point, and *around advice* run in place of their join point. For example the advice

```
before() : foo
{
  System.out.println ("After pointcut foo");
}
```

prints out a simple message when the pointcut `foo` is reached and before the computation of the original code proceeds.

Pointcuts are able to expose the execution context at their join point. This context information can be used in advices.

```
pointcut setX(ClassA a, int x) :
{
  call (void ClassA.setX(int))
    && target (a)
    && args (x);
  before(ClassA a, int x)
  {
    System.out.println ("New value for x: " + x);
  }
}
```

This pointcut `setX` exposes the two values from calls to method `setX` of `ClassA`: the instance of `ClassA` that receives the call and the new value for `x`. The advice prints out the new value of `x` before the `setX` method is invoked.

Introductions are used to modify classes and the hierarchy by adding new members and changing relationships between classes. Introductions change the declarations of classes. Since these changes are inherited they effect the rest of the program. Introductions are static, i.e., they take place at compile time whereas advices operate during runtime.

The definition of *aspects* is very similar to classes. Aspects define the units for implementing cross-cutting concerns using pointcuts, advices, and introductions.

```
aspect Count
{
  private int count = 0;
  pointcut CountSetX () : call (ClassA.setX (int));
  before() : CountSetX ()
  {
    count++;
  }
}
```

The aspect `Count` introduces `count` as a new member of `ClassA` and defines a pointcut for the `setX` method of `ClassA`. The before advice increments the newly introduced `count` variable every time the `setX` method is invoked.

3 Monitoring Distributed Applications with AOP

Managing distributed systems requires knowledge about the status of the constituting components. Their status can be deduced from information gained by monitoring the components. This is realised by inserting instrumentation code into the managed system. Here, two basic approaches can be distinguished: First, an intrinsic instrumentation approach where code is inserted into the component under monitoring. Second, an extrinsic instrumentation approach which uses additional components, either hardware or software to monitor components of the system, e.g., by analysing the log files of a Web server. Typically, the intrinsic approach is able to provide more fine-grained data, since component internals can be used. The extrinsic approach typically provides more coarse-grained information since it has to rely on information exported by the component. However, the latter approach is less intrusive.

Our approach based on aspect-oriented programming can be characterised as an intrinsic instrumentation approach since it inserts the instrumentation into the code of the component. In the following sections we describe how our approach helps to make components more manageable.

3.1 Measuring Servlet Response Times Using ARM

In the area of Service Level Management, Service Level Objectives (SLOs) defining availability criteria and performance-related metrics, e.g., response times, are the key issues of many Service Level Agreements. The fulfilment of an SLA has to be monitored to prove compliance with the defined SLOs. In the terms of AOP, response time is a non-functional requirement and its measurement a cross-cutting concern since many parts of a component are involved. Even further, several components may be involved and their measurements have to be correlated.

For response time measurements the ARM API is a well accepted approach for instrumenting applications at the source code level. The API supports execution time measurements of source code fragments termed ARM transactions within a distributed application. ARM provides ways of correlating nested measurements, even across host boundaries. For this purpose the API provides correlators that identify ARM transactions. Correlators can be supplied on creating a nested transaction for relating this to the enclosing transaction. However, passing of correlators between application components, which might prove difficult especially in distributed systems, is the task of the application developer.

3.2 Scenario

In the future, application service providers will offer a wide range of different services from simple web hosting to complex e-business applications. These complex applications are typically realised in a Web-based e-business environment consisting of a Web server as central entry point and a web container to provide Java server pages and servlets. Most of the business logic is implemented by Enterprise Java Beans (EJB) that live in an EJB container. CORBA is used to implement business logic as well as an integration middleware for legacy components. A relational database ensures the persistent storage of the enterprise data.

Our previous research has revealed that instrumenting such a complex environment using ARM is a difficult task [17]. Since the ARM specification does only specify the format of the ARM correlator which is used for correlating nested transactions, but not the mechanism to transfer them between process boundaries, this task is up to the software developer.

Infrastructure components like a Web server and a Web container can be instrumented transparently for the application developer. However, application components, such as Servlets, should also be instrumented transparently for the application developer in order to ensure cost effectiveness. In addition, an automatic instrumentation guarantees a consistent instrumentation of all application components which is the prerequisite of comparable measurements. The following section describes our aspect-oriented instrumentation for transparently measuring Servlet response times using the ARM API.

3.3 Aspect-Oriented Instrumentation Approach

A typical task of a Servlet is the retrieval of records from a database system. Thus, the duration of a database query is an important unit of work to be measured. Listing 3.1 shows the code of a simple Servlet that queries a number of records from a database and subsequently transforms the result set into a Web page. For the clarity of the code all `try` and `catch` clauses are not shown in the listing.

The `init` method of the Servlet, which is executed only once during the initialisation of the Servlet, is responsible for setting up the connection to the database. The `destroy` method closes the database connection when the Servlet is destroyed. The database query is performed as part of the `doGet` method. Finally, the results of the query are transformed into HTML and sent back to the client.

An application developer who has to instrument the Servlet with ARM manually would place code for initialising the ARM environment into the `init` method, and code for shutting down the ARM environment into the `destroy` method. Within the `doGet` method the application developer has to check if a parent correlator was handed over to the Servlet. Afterwards, a new ARM transaction must be created and appropriate start and end points for the measurements have to be identified in the application code. In our example, start and stop commands were placed around the execution of the database query.

Listing 3.1 Servlet for querying a database

```
public class MyDB extends HttpServlet {
  final String url = "jdbc:mysql://dbhost:3306/mysql";
  final String driver = "com.mysql.jdbc.Driver";
  final String query = "select host, user from user";
  Connection conn;

  public void init(ServletConfig config) throws ServletException {
    super.init(config);
    Class.forName(driver);
    conn = DriverManager.getConnection(url, "dbuser", "");
    conn.setReadOnly(true);
  }

  public void destroy() {
    if(conn != null && !conn.isClosed())
    conn.close();
  }

  public void doGet(HttpServletRequest request, HttpServletResponse response) throws ... {
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(query);
    java.lang.Thread.sleep (20);
    response.setContentType("text/html; charset=ISO-8859-1");
    PrintWriter out = response.getWriter();
    out.println("<HTML><BODY>");
    ...
    out.println("</BODY></HTML>");
    rs.close();
    stmt.close();
  }
}
```

In this simple example the manual instrumentation is considerably simple. Nevertheless, the application developer has to handle all the complexity of the ARM environment in order to achieve useful measurements, i.e., the application developer has to understand the business requirements of the application to correctly implement the functionality as well as the management requirements to support the management of the application component later on. In addition, the manual instrumentation leads to enormous costs since many code pieces have to be instrumented. This work is monotonous and distracts from implementing the business logic. Using inheritance does not really solve the problem since it mainly simplifies the initialisation and destruction of the Servlet. Furthermore, it requires a refactoring of the existing code which is again a potential source of errors.

The aspect-oriented approach does not require modifications of the existing code. Here, the instrumentation code is encapsulated as an aspect which may be done by a different developer who is familiar with ARM environments, while the application developer can concentrate on the application logic. The application code is simply recompiled using a special compiler, the aspect weaver, which connects the aspect code with the application code. Thus, instrumentation can also easily be integrated into an existing application.

Listing 3.2 depicts the aspect code for measuring the duration of database queries. (Again, `try` and `catch` clauses are not shown in the listing.) First, a number of local variables are defined that are used within the aspect for the ARM measurements. Thereafter, the first pointcut identifies the `init` method of the Servlet, the second pointcut identifies the `doGet` Servlet method, and the third pointcut identifies all invocations of the `executeQuery` method of the

Listing 3.2 Aspect for instrumenting JDBC calls with ARM

```
aspect MyDBAspect {
  ArmTransactionFactory tranFactory;
  ArmTransaction dbTransaction;
  HttpServletRequest request;
  byte[] myByteUuid = null;

  pointcut arm_init() : call(void *.init(..));

  pointcut arm_doGet(HttpServletRequest request, HttpServletResponse response)
    : call(void *.doGet(HttpServletRequest, HttpServletResponse))
      && args(request, response);

  pointcut execQuery(String content)
    :  call(ResultSet Statement.executeQuery(String)) && args(content);

  void around() : arm_init() {
    Class tranFactoryClass;
    tranFactoryClass = Class.forName(tranFactoryName);
    tranFactory = (ArmTransactionFactory)tranFactoryClass.newInstance();
    myByteUuid = new byte[] { (byte)0x6c, ..., (byte)0xf1 };
    proceed();
  }

  void around(HttpServletRequest request, HttpServletResponse response)
    : arm_doGet(request, response)  {
    ArmUUID uuidDbTransaction;
    this.request = request;
    uuidDbTransaction = tranFactory.newArmUUID(myByteUuid);
    dbTransaction = tranFactory.newArmTransaction(uuidDbTransaction);
    proceed(request, response);
  }

  before(String content): execQuery(content) {
      dbTransaction.start((ArmCorrelator)request.getAttribute("CORRELATOR"));
      ArmCorrelator corr = dbTransaction.getCorr();
  }

  after(String content) returning():execQuery(content) {
      dbTransaction.stop(ArmConstants.ARM_GOOD);
  }
}
```

`Statement` class. The second and the third pointcut also expose variables from their context.

The instrumentation code of the aspect is defined in its advices. The `arm_init` advice is an around advice that traps the execution of its join point (`init` method of Servlet), i.e., the code of the advice will be executed in place of the original code. By including the `proceed` statement at the end of the advice the original code of the `init` method will also be executed. The advice initialises the ARM environment.

The `arm_doGet` advice traps the execution of `doGet` Servlet method and handles the initialisation of a new ARM transaction for measurement. By using the `proceed` statement the original code is executed.

The ARM measurements are performed by using a `before` and an `after` advice for the `execQuery` pointcut. These advices place ARM start and stop statements around the execution of the database query. When starting a measurement the instrumentation code tries to extract a parent correlator from the `CORRELATOR` attribute of the the request object. This parent correlator will then be used as basis for the measurements within the Servlet and later enables a management application to correlate measurements of different components.

4 Performance Evaluation

To evaluate the efficiency of the aspect-oriented instrumentation approach we performed a series of measurements under lab conditions. The goal was to determine the runtime overhead caused by the aspect-oriented instrumentation approach.

Our prototype involved the Tomcat Web container in version 3.21 which was instrumented using the tang-IT ARM library [18]. The Servlet code was developed using the Sun Java Development Kit (JDK) 1.4.0; we used AspectJ Version 1.0.6 as aspect-oriented programming environment. The experiments were performed on a AMD Athlon XP 1880+ with 512 MB RAM running SuSE Linux 7.3 with Kernel version 2.4.16. The Servlet performed a query on a local MySQL database in version 3.23.55.

The measurements consisted of a client sending 1000 consecutive requests to the Servlet. The think time between two requests was 100 ms. Three independent experiments were performed using a uninstrumented Servlet, a manually instrumented Servlet, and a Servlet instrumented using our aspect-oriented approach. The results of the measurements are shown in figure 1. The values are based on measuring the overall processing time of the Servlet within the Tomcat Web container.

At first glance, the overhead of the instrumentation (manual and aspect-oriented) is quite high especially for performance measurements. The overhead is caused by the additional instrumentation code that was inserted into the Servlet. Since the functionality of the Servlet is minimal, as indicated by the mean response time of about 1 ms, the execution time proportion of the instrumentation code is considerably high. Therefore, the measurements can be regarded as representing the worst case. In real world Servlets with more complex application logic, the overhead will be considerably smaller. The overhead of the aspect-oriented instrumentation is slightly higher than the manual instrumentation. However, the aspect-oriented instrumentation offers a number of advantages such as cost effectiveness, no code pollution of application code, and separation of concerns. These advantages outweigh the slightly higher overhead.

We also determined the overhead from an end-user perspective, i.e., the response time as seen by the client (see figure 2). Here, the overhead of the instrumentation code went down to about 7% for manual instrumentation and about 8% for the aspect-oriented instrumentation. This overhead is within acceptable

	(1) without instrumentation	(2) with manual instrumentation	(3) with aspects
mean (ms)	1.145	1.473	1.536
std. dev. (ms)	0.954	1.247	1.274
% increase		28.646 %	34.148 %

Fig. 1. Processing times of the Servlet code

	(1) without instrumentation	(2) with manual instrumentation	(3) with aspects
mean (ms)	4.059	4.176	4.426
% increase		7.203 %	8.063 %

Fig. 2. Client side response times

boundaries for performance measurements. Again, these values reflect a worst case scenario, since the Servlet contains only minimal application functionality.

The results of the performance evaluation show that the aspect-oriented approach is very appropriate for performance instrumentation since its overhead is only slightly higher than the overhead caused by a manual instrumentation. The higher overhead of the aspect-oriented approach is due to the fact that it involves more instrumentation code than the manual instrumentation. The AspectJ environment generates a class representing the code of the aspect. In the class that has to be instrumented, hooks to this aspect class are added.

In principle, the number of measurement points within a component should be as minimal as necessary in order to keep the overhead as low as possible. Alternatively, the implementation of the ARM library and the AOP environment could be optimised which would require modifications of their source code. This was not the focus of our work.

5 Conclusions and Future Work

The increasing importance of Service Level Management implies a strong demand for management instrumentation of application components.

In this paper, we presented an aspect-oriented approach for instrumenting application components. The instrumentation code is encapsulated as an aspect and woven into the application code during compile time. Except for the use of a different compiler, the aspect weaver, the instrumentation process is completely transparent to the application developer. Thus, he is relieved from the burden of manual instrumentation and can concentrate on the application logic. We achieve separation of concerns by decoupling the application logic from the management logic. Thus, the efficiency of the overall software development process is increased, since separate concerns can be treated in separate tasks performed by different experts. An additional strong advantage of our approach is, that the aspect code can be reused in other applications. This is usually impossible in conventional instrumentation approaches.

For demonstrating our approach we chose the performance instrumentation of a Servlet in a typical e-business application scenario. Concretely, the duration of JDBC database queries was measured using the ARM API. The efficiency of the aspect-oriented instrumentation is demonstrated by a series of measurements. The instrumentation approach has been compared to an uninstrumented Servlet and a manually instrumented Servlet. The results show that in a worst case scenario the overhead of the aspect-oriented approach is only slightly higher

than the overhead of a manual instrumentation. This demonstrates that clarity of program structure and support for code reuse can be achieved without a substantial loss of efficiency.

Since our aspect-oriented instrumentation approach has demonstrated its viability and effectiveness, we are optimistic to apply the approach successfully to other management problems as well. So far, we also used AOP for instrumenting Web Services with ARM. There we chose a dual approach, i.e., we manually instrumented the Web Services platform and used AOP to instrument the Web Service itself. The results are comparable to the results published in this paper. Our future work will concentrate on using AOP to tackle management problems other than ARM instrumentation, e.g., we are working on a transparent integration of JMX instrumentation into EJBs. We expect to gain general insights into the use of AOP in management applications and to understand the limitations of the approach.

Acknowledgements

The authors like to express their gratitude to Alexander Hoffmann, member of the Distributed Systems Lab at Fachhochschule Wiesbaden - University of Applied Sciences, Germany, for his helpful discussions and implementation work.

References

[1] Lewis, L.: Managing Business and Service Networks. Kluwer Academic Publishers (2001) 209

[2] Sturm, R., Morris, W., Jander, M.: Foundations of Service Level Management. SAMS Publishing (2000) 209

[3] Verma, D.: Supporting Service Level Agreements on IP Networks. Macmillan Technical Publishing (1999) 209

[4] Katchabaw, M. K., Howard, S. L., Lutfiyya, H. L., Marshall, A. D., Bauer, M. A.: Making Distributed Applications Manageable Through Instrumentation. In: 2^{nd}Second International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97). (1997) 210

[5] Mehl, O., Becker, M., Köppel, A., Paul, P., Zimmermann, D., Abeck, S.: A Management-Aware Software Development Process Using Design Patterns. In: 8^{th} IFIP/IEEE International Symposium on Integrated Network Management (IM 03). (2003) 579–592 Colorado Springs, USA 210

[6] Distributed Management Task Force: Common Information Model (CIM) Specification. (1999) Version 2.2. 210

[7] Debusmann, M., Schmid, M., Kröger, R.: Generic Performance Instrumentation of EJB Applications for Service-Level Management. In Stadler, R., Ulema, M., eds.: 8^{th} IEEE/IFIP Network Operations and Management Symposium (NOMS). (2002) Florence, Italy 210

[8] The Open Group: Systems Management: Application Response Measurement (ARM). (1998) Open Group Technical Standard, Document Number: C807 210

[9] Sun Microsystems, Inc.: The Java Management Extensions Instrumentation and Agent Specification, v1.0. (2000) 210

[10] Debusmann, M., Schmid, M., Kröger, R.: Measuring End-to-End Performance of CORBA Applications using a Generic Instrumentation Approach. In: 7^{th} IEEE Symposium on Computers and Communications. (2002) Taormina/Giardini Naxos, Italy 210
[11] Hauck, R.: Architecture for an Automated Management Instrumentation of Component Based Applications. In: 12^{th} International Workshop on Distributed Systems: Operations & Management (DSOM'2001). (2001) 210
[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J. M., Irwin, J.: Aspect-Oriented Programming. In: European Conference on Object-Oriented Programming (ECOOP), Springer (1997) 211
[13] Becker, C., Geihs, K.: Generic QoS Support for CORBA. In: 5^{th} International Symposium on Computers and Communications (ISCC 2000), Antibes, France, Springer (2000) 211
[14] Hauck, F. J., Becker, U., Geier, M., Meier, E., Rastofer, U., Steckermeier, M.: AspectIX: A quality-aware, object-based Middleware Architecture. In: New Developments in Distributed Applications and Interoperable Systems (DAIS'01), Kluwer (2001) 115–120 211
[15] The AspectJ Team: AspectJ. Xerox Corporation. (2002) http://www.eclipse.org/aspectj/ 211
[16] The AspectJ Team: The AspectJ Programming Guide. Xerox Corporation. (2002) http://download.eclipse.org/technology/ajdt/aspectj-docs-1.0.6.tgz 211
[17] Debusmann, M., Schmid, M., Schmidt, M., Kröger, R.: Measuring Service Level Objectives in a complex Web-based e-Business Environment. In: 10^{th} HP OpenView University Association Workshop (HPOVUA). (2003) Geneva, Switzerland 214
[18] tang-IT Consulting GmbH: tang-IT Application Response Measurement (ARM). (2003) http://arm.tang-it.com/ 217