Appraoches to Adaptive Middleware

Sushil K Bajracharya
sbajrach@ics.uci.edu

Abstract

Aspect Oriented Programming (AOP) has emerged as new paradigm for separating cross-cutting concerns in software. With AOP software developers get a more expressive platform where they can represent and work with common *aspects* of the system that crosscut more than one software module. Reflective middlware on the other hand provide a platform where the cores aspects of Distributed Systems such as Remote Creation, Directroy Services and Distributed snapshots can be respresented and modeled as meta-level entities. With such a two level separation we get a flexible model of a distributed middlware that can provide dynamic configuration of middleware components hence giving a Adaptive framework making runtime dynamism of components possible as demanded by new generation of ubiquitous and pervasive computing environments. With a wide spread adoption of AOP in different domains and with the advent of Dynamic AOP frameworks recent advances have been made to come up with AOP based middleware that leverage the underlying concepts of AOP and reflective middleware and provide a novel environment to develop dynamic and adaptive middleware.

This survey paper presents a general background on reflective and adaptive middleware, AOP and dynamic AOP and present some of the interesting research works related to reflective and AOP based adaptive middleware.

1 Inroduction

The meaning of middleware in today's context can be broadly construed. Middleware span from software system providing services to clusters of computers forming a supercomputing environment to software serving embedded devices making the ubiquitous computing experience of an end user as efficient as possible. In order to address the demanding computing needs by the proliferation of pervasive computing platforms systems that can adapt to the changing environment on-the-fly or during runtime has given rise to adaptive middleware architectures that support dynamic plug-and-play of components and thus functionalities they provide to the applications that demand such facilities and that rely on them. Reflective middleware have been taken as one of the approaches in building such breed of middleware platforms and more recently AOP and closely related technologies like *Adaptive Programming/Aspectual Components* has been used as a new style of harnessing such reflective architectures.

We start in this survey paper by looking into some of the recent works that had been done in adaptive and dynamic middleware especially looking into the case of reflective middleware, then we present the general background of AOP and discuss in particular dynamic AOP, some of the recent industry and research initiatives to incorporate AOP as a middleware developing tool will be discussed and then finally we discuss a particular example of AOP being used in developing a dynamic middleware for managing web services.

2 Middleware - A platform for interoperability and adaptability

One way of looking at middleware is to consider it a reusable layer of software that is responsible to provide end-to-end application functionality where the applications might run on heterogenous environments using diverse software components. As such a layer middleware come in between the application and operating systems/network protocol layer. Along with providing such interoperability services middleware can also serve as a layer that provides Qos guarantees and other non-functional services to applications running on top of them. These requirements become evident in application domains like Distributed Real-time and Embedded Systems [Sch] and, Ubiquitous and Multimedia applications [RCK01].

2.1 Adaptive middleware

Adaptive middleware can change its functional and non-functional constraints like Qos-related properties either *statically* or *dynamically*. Some examples of middleware that belong to this category are the *Quo* middleware [Sch] and *The ACE ORB* (TAO). Static adaptation allows leveraging the capabilites of specific platform or environment the middleware runs in and dynamic adaptation as the name suggests allows application to run with optimum system responses where the requirements or environment itself is changing. Reflective middleware described next can be used as a middleware platform for dynamic adaptability.

2.2 Reflective middleware

Reflective middleware build upon the concept of *Computational Reflection*. A Reflective system is the one that supports an associated *causally connected self represenatation* (CCSR). Reflective middleware is a middleware system that provides *inspection* and *adaptation* of its behavior through an appropriate CCSR. [Cou]. With this capability the reflective middleware model can provide a principled and efficient way of dealing with highly dynamic environments that supports development of flexible and adaptive systems and applications [KCBC02]. Simply put reflective middleware has the capability to *reason about and act upon itself*. Inspection allows the current state of the system to be observed and adaptation allows the system's behaviour to be changed at run time.

The major features of a reflective middleware can be highlighted as follows -

1. A reflective middleware is implemented as a group of collaborating components. It an open model as opposed to a black-box view of a monolithic system. This makes it possible to build lightweight middleware platforms with small resource footprints by selecting and composing only those components of interest.
2. There is a support for dynamic customization of such components and they are controllable at fine-grain level through meta-level interfaces.
3. Overall, a reflective midddleware gives a two-level view of the system. At the base level are the application objects where as the the meta-level deals with the reflective computation. The meta level exposes the internal repesentation of middleware components as entities that can be manipulated at runtime. There is a causal connection between the the objects that run at base level and components at the meta-level.

Many pervasive applications today are need in of such a adaptive operative platform as provided by reflective middleware. For example, in today's interconnected world a user of a typical internet based applcation can be accessing the application from various types of devices - a powerful workstation at home, notebook while driving, a PDA or even a wrist watch. The user might switch between these or even using the same device the environment might change (for instance network bandwidth might get better/worse). To provide an optimum user experience a statically configured system will surely outperform even if it has all the modules to meet the needs of all the different usage scenarios, in such case a middleware solution that can dynamically configure an application to adapt to changing application requirements will be a viable solution. [RCK01] discusses some of these issues that can be addressed by reflective middleware. [KCBC02] presents *Dynamic TAO* and *OpenORB* as two examples of reflective middleware and also lists some of the other implementations such as - *OpenCORBA*, *Quarterware* and *mChaRM*. [NWP] explains how adaptive and reflective middleware system can be used to implement a CCM (CORBA Complaint Model) - complaint object request broker (ORB) using the *TAO* ORB system.

We will look into the innards of a reflective middleware *ComPOSE|Q* in section 3. Before that we discuss some of the issues that arise in these systems in the following sections.

2.2.1 Composability in Reflective Middleware

With the exciting features reflective middleware provides there comes the issue of composability of various services in such middleware. As such services can be dynamically and freely composed to achieve desired functionality, a necessity of validating such compositions against several critical parameters arises. With more flexibility comes the possibility of more conflicts and configurations that can violate critical safeguards. Conflicting issues and services like concurrency, garbage collection and process migration has to be properly looked into.

The changing, often complex, interactions among distributed policies and services within the middleware layer can violate correctness, destroy validity, and alter the semantics of applications built above generic distributed management infrastructures [Ven02]. A solution to this problem is to have a well designed meta-architectural framework and a formal basis for such a framework [GNV], [Ven02], [VT95].

2.2.2 Formal basis for safe composability

The Two Level Actor Machine (TLAM) model [VT95], [VT01] is an example of a meta-architectural framework that has been used to provide a formal semantic basis for specifying and reasoning about the properties of and interactions among middleware components in a general purpose middleware framework - CompOSE|Q. TLAM is based on the *Actor* model, it is a two level Actor systems with actors distributed over network. The system is made up of two kinds of actors - *base* and *meta* actors. Base actors are application objects and meta actors are part of the runtime system. TLAM provides a formal description of three basic core system services i) Remote Creation , ii) Distributed Snapshot and iii) Directory Services. Further higher level services can be defined from these three core services.

TLAM can be used as a model to ensure safe composability of services in open distributed system as it allows a formal specification and checking of non-interference properties such as -

1. Interference between actors with a common acquaintance.

2. Interference between base and meta level actors on the same node.

3. Interference is between meta actors implementing different services and thus modifying base level actors in possibly incompatible ways.

3 CompOSE|Q - A Case Study of Reflective Middleware

CompOSE|Q (Composable Open Software Environment with QoS) [WEBa], [ea] is a Qos enabled customizable middleware framework for distributed computing based on the TLAM model as described in section 2.2.2. CompOSE|Q allows the concurrent execution of multiple resource management policies in a distributed system in a safe and correct manner [ea].

Since based on TLAM, CompOSE|Q consists of actors (base and meta) that are distributed over network, run parallel to each other and can communicate with each other via asynchronous message passing. It is an actor based framework that permits customization of resource management mechanisms such as placement, scheduling and synchronization.

The architecture of CompOSE|Q is made up of three components -

1. Modules implementing the three core services - remote creation, distributed snapshot and directory services with interaction constraints that ensure their concurrent execution with each other and other meta level services.

2. Common services built upon the core services such as actor migration, replication of services and data, actor scheduling, distributed garbage collection, name services etc with their own definitions and interaction constraints.

3. Qos specification and enforcement mechanisms.

CompOSE|Q includes a set of *meta level resource management services* to implement sophisticated policies and mechanisms for QoS management. It also includes a *reflective communication service architecture* which basically extends TLAM with a *composable reflective communication framework* (CRCF). This is the core that provides correct composition of communication services to QoS-based applications in a transparent and scalable fashion while ensuring correctness of basic middleware services. The *CompOSE|Q Runtime Architecture* is implemented in Java and is made up of three basic components - i) A *NodeManager* that manages and coordinates various components on a node, ii) A *NodeInfoManager* that manages information needed by the local actors and interfaces with the directory service and iii) A *communication sub-system* that handles messaging between actors.

4 AOP for Dynamic Middleware

So far we have discussed how reflective middleware has been exploited as a vehicle for separation of concerns in distributed middleware systems. Now we look into the other side of the spectrum where we discuss the state-of-the-art technique in separation of concerns in software development with a new programming style called Aspect Oriented Programming (AOP). Later we will see how this technology is providing solutions to solve some the similar issues that reflective middleware is trying to address.

4.1 Separation of Concerns with AOP

Even with the traditionally established design principles such as modular decomposition, abstraction and information hiding there remain subtle implementation issues that bind the modular components of a software together as a cohesive whole. These issues (such as concurrency and memory access patterns [KLM+97], especially perceptible during runtime) are not prone to the conventional decomposition techniques. These issues *crosscut* software components and stand out as the *properties that affect the performance or semantics of the components in systematic ways* [KLM+97]. AOP brings into light a new perspective of problem decomposition (or, arguably composition) and software organization to address such cross cutting issues.

The principal goal of AOP is to provide mechanisms to allow separation of crosscutting *aspects* from *components* in a system. Aspects are the issues that cannot be cleanly encapsulated as a program unit whereas components can be modelled as separate logical entities (like classes in object oriented programs). This separation results in an easy to understand and an efficient program. To achieve this, the framework for AOP comprises of following components -

Component language A high level language with constructs like procedures, classes etc to program the components. Component language can be transformed to produce *component program*.

Aspect Language An extension to the component language or a separate language to program the aspects. Aspect language gives *aspect program* when implemented.

Aspect weaver This is a tool that processes the component and the aspect language (precisely the programs written in these languages) and combines them together to produce the final executable code.

Few important points should be noted regarding this framework for AOP -

- Component programs are simple as they cannot *preempt* anything aspect programs need to control. For example, if Java is being used as a component program and if concurrency is one of the aspects then using keywords like `synchronized` and methods like `wait()` should not be allowed in the component language. This should rather be addressed by the aspect language.
- The aspect weaver does not work as an optimizing compiler in producing efficient code by combining the component and aspect language. All the possibilities for optimizations should be already represented by the aspect language in the aspect program. The weaver simply integrates programs written in component and aspect languages.
- Weavers take into account the presence of *join points* during the *weaving* process. Join points are elements of the component language semantics that the aspect programs coordinate with. A simplest example might be the point of method call in a program or a point in a program that can be captured to implement a technique like loop fusion.

AOP is similar and related to other advanced separation of concerns techniques like *Adpative Programming* [Lie96], *Metalevel Programming* [Kic92] [GKB91], *Composition Filters* [Ber94] and *Subjective Programming* [AWB+93] [HO93].

4.2 Staic Vs Dynamic AOP

Since the first implementation of a generic version of AOP platform AspectJ was out, there have been tremendous efforts and alternative strategies to realize AOP for separating crosscutting concerns in software. Current AOP styles can be categorized into two groups -

Static AOP Frameworks like AspectJ [WEBb] fall into this category. Under these frameworks aspects are just a more expressive representation of programming constructs and after weaving and generation of the target code the aspects are lost.

There has been some efforts in the middleware community that demonstrate that rewriting or adopting middleware aspects using static AOP techniques provide a higher level of separation of concerns as it helps to abstract out many crosscutting issues in distributed middlware systems [SG], [HCG], [ZJ03].

Dynamic AOP These frameworks unlike static AOP maintain the aspects either till the load time or fully during the runtime. Those providing full dynamic AOP capabilities provide an aspect language of some kind, a weaver and a reflective platform capable of maintaining the meta-model of the aspects allowing them to treat as first-class-objects during runtime. AOP frameworks like JAC (Java Aspectual Components) [RPM], PROSE () [PGA01] and JAsCo [SVJ03] fall under this category. Frameworks like JBoss [WEBc] support load time waeving. All of these AOP tools are based on the Java programming language. Javassist [WEBd] is a popular MOP implementation and BCEL (Byte Code Engineering Library) is a widely used bytecode manipulation library used to build dynamic AOP layers in tools like these.

Some of the recent works that support dynamic remodularization of components like *Aspectual Components* [LLM99], [MO02] also are based on principles of AOP. Along with these, many of the approaches and facilities provided by the dynamic AOP solutions are similar to those as provided by reflective middlware. In the following section we discuss AspectJ as a static AOP tool, JBoss as an AOP based middlware solution with load time weaving and JAsCo as a dynamic AOP framework that has capabilities to be used as an adaptive middlware solution.

4.2.1 AspectJ

In a nutshell the current version of AOP is comprised of the following elements.

Component Language Since its conception AspectJ has been using the Java programming language as its component (or base) language and no change has been made in this till today.

Aspect Language The aspect language named as AspectJ itself is what adds the expressive capability to the component language Java.

The AspectJ (aspect language) can be described in terms of the following constructs that constitute it -

Join points AspectJ now comes with a more powerful join point model that describes several join points besides method calls, reception or execution. As mentioned earlier, join point is a well defined point in an execution of a program. The strength of an AOP framework is governed by how fine grained its join point model is. AspectJ does not allow an arbitrary representation of the cross-cutting issues in a program as *aspects*. In fact only those issues that cross-cut the join points defined in AspectJ can be modelled as *aspects*.

[WEBe] lists eleven join points for the latest AspectJ release and describes all the join points and their semantics in greater detail. Some of them are **method call**, **constructor call**, **constructor execution**, **field reference**, **field set** and **handler execution**.

pointcuts	meaning
call (*Foo.new(..)*)	a **call** to any constructor of *Foo*
execution(* *Foo.*(..) throws IOException*)	the **execution** of any method of *Foo* that is declared to throw *IOException* (note the use of wildcard * here)
set (*!private * Point.**)	when *any non-private* (implied by *!*) field of *Point* is **assigned**
handler (*IOException+*)	when an *IOException* or its *subtype* (implied by +)is **handled** with a catch block

Table 1: Some of the pointcuts available in AspectJ

Pointcuts Pointcuts are constructs in AspectJ that selectively allow to choose certain join points in the program flow. AspectJ defines a set of seventeen primitive pointcuts that allow all the basic join points to be captured. Furthermore, new pointcuts can be defined on the basis of old pointcuts, and pointcuts allow use of wildcards and logical operators like *&&* (meaning logical *and*), *!* (meaning logical *not*) and || (meaning logical *or*) while selecting and specifying join points.

Table 1 lists few examples of pointcuts in AspectJ. It should be noted that **call**, **execution**, **set** and **handler** are the base pointcuts that AspectJ defines.

Advices Advices define a body of code that is executed when a pointcut is reached. An Advice binds together a pointcut (that pick up the join points) and a body of code (Java code) to run at each of those join points.

AspectJ defines three kinds of advices-

1. *Before*: Before advices run upon reaching the join points and before the program continues with the join points.
2. *After*: After advices run upon reaching the join points and after the program continues with the join points.
3. *Around*: Around advice runs in place of the join point it operates over, rather than before or after it.

Aspects Aspects are special programming constructs that cleanly encapsulate the cross cutting concerns in separate units. They are similar to the notion of classes in the component program. Aspects are combinations of pointcuts and advices.

Most of the AOP implementations today show a similar kind of constituent elements as AspectJ.

4.2.2 JBoss

JBoss AOP framework [WEBc] comes as a part of the JBoss Java Application Server [WEBf]. JBoss AOP can be used as a standalone AOP framework as well as it the WSML as it represents the collection of aspects that represe significant benefits of JBoss AOP using with the JBoss server is that JBoss server ships its system-level services such as Transactions, Remoting, Clustered Remoting and Transactional Locking as aspects that are written on top of the framework.

Comparing JBoss AOP and AspectJ The significant differences between JBoss AOP and AspectJ can be listed as follows -

- JBoss primarily uses XML as its aspect language to model AOP constructs like Introductions, and pointcuts.
- JBoss uses interceptors (or interceptor classes) that work as the advice construct in AspectJ. Interceptor classes can be used to intercept method invocations, constructor invocations and field access.
- JBoss AOP adds the possibility of declarative programming in Java by allowing metadata to be defined as aspects in XML files that can be used to add attributes to classes and other elements of Java.
- JBoss AOP uses bytecode manipulation to attach the interceptors to the component code and uses its own class loader as it does runtime weaving.
- JBoss provides a tool called AOP Management Console with web based interface that allows to view classes that have been loaded by its classloader and being instrumented in the AOP framework. It even shows a static trace of interceptor chains, introductions, interceptor pointcuts and metadata attached to the loaded classes.
- Above all, the major difference between the JBoss AOP and AspectJ lies in the weaving process. AspectJ can be viewed as a statically weaving AOP framework that either takes Java sources or complied classes and combine them together giving classes that can be loaded and executed in the Java Virtual Machine. With JBoss the process is more dynamic. JBoss AOP uses its own class loader to combine the Java code and instrument it by reading the XML representation of the aspect code. Then it uses a meta-object library called *Javassist* [WEBd] to do load time weaving of aspects.

5 JAsCo - A Case Study in AOP based Adaptive Middleware

5.1 JAsCo AOP

JAsCo basically is an AOP implemenation language/framework. It is closely related to and borrows concepts from *Aspectual Components* [LLM99]. The JAsCo langage introduces two concepts: *aspect beans* and *connectors*. An aspect bean describes behavior that interferes with the execution of a component by using a special kind of inner class, called a *hook*. A *connector* is used for deploying one or more hooks within a specific context. *Aspects beans* are used for describing those functionality that crosscuts several components specified in the connectors that deploy the redirection aspect. Several connectors can exist each in charge of deploying the redirectio class-members. Hooks can be used to specify when the normal execution of a method of a component should be intercepted and what extra behavior should be executed.The *connector* on the other hand contains three kinds of constructs: one or more hook initialization (identical to a Java class instantiation), zero or more behavior method executions, and finally any number of regular Java constructs. In summary the aspect beans specify when the normal execution of a method should be intercepted and what extra behaviour should be executed and teh connectors specify where the crosscutting behaviour should be deployed.

JAsCo has support for advanced aspect combinations. For example, it is flexible enough to specify combinations beyond simple composition rules such as when aspect A is applied aspect B can not be applied. For this JAsCo introduces a more flexible and extensible system that allows to define a combination strategy

using regular Java. A JAsCo combination strategy works like a filter on the list of hooks that are applicable at a certain point in the execution. This feature some what provides a kind of composition rule that allows to compose related croscutting behavior without avoiding conflicts.

The JAsCo language itself is implemented as a new *aspect-enabled component model*, which contains built-in traps that enable to interfere with the normal execution of a component. The JAsCo component model is backward-compatible with the Java Beans component model. Furthermore, the JAsCo component model allows very flexible aspect application, adaptation and removal at run-time. [SVJ03]

The approach JAsCo takes makes it very flexible to support unanticipated run-time changes. Connectors can be loaded and un-loaded at runtime. It implements a special registry called the *connector registry* that detects whether connectors are removed or added to the system and that can take appropriate actions. It also support the instantiation of a hook on expressions that contain wildcards that are matched at run-time so that when a new component is added to an application, it is automatically affected by all aspects that were declared using such wildcards.

5.2 WSML - an Adaptive middleware implementation using JAsCo

Web services has been emerging as a new solution to distributed computing that allows to put together loosely coupled pieces of software services together to buld applications. However, heterogenity and interoperability is still a problem in web services application domain. Web Services Management Layer (WSML) is a middleware layer built using JAsCo AOP that allows decoupling web services from the core application. It realises the concept of just-in-time integration of services where multiple services or compositions of services can be used to provide the same functionality. [VCJ] describes the architectre and implementation issues of WSML in much greter details but in brief it has the following elements -

Abstract Service Interfaces (ASI) enables web service requests to be formulated in an abstract way from the client application (through WSML layer) such that the WSML will be responsible for making the translation to a concrete service one that is semantically equivalent to that as requested.

Service Oriented Aspects This forms an essential core of the WSML as it represents the collection of aspects that represent the management services that WSML provides. One such aspect is the *redirection aspect* that is responsible to make the abstraction provided by the ASI realisable we need to provide means to map the generic description to concrete service invocations. Redirection aspect is in charge of redirecting the generic requests to the concrete services that will provide the functionality required. The mapping to concrete web services is specified in the connectors that deploy the redirection aspect. Several connectors can exist each in charge of deploying the redirection to a concrete web service. Other aspects that belong to this part of the WSML core might be related to client side service management such as accounting or billing issues for web-service usage.

Service Selection and Monitoring Since WSML allows ASIs to be mapped to concrete services and service compositions. This mechanism enables the layer to decide which semantically equivalent services will be effectively chosen when the application invokes certain functionality. This is achieved by means of dynamically selecting and activating the corresponding service connectors. The *selection* module is responsible for regulating the activation of connectors. In order to control the properties of the web services that influence the decision, *monitoring* is required. Service Selection and Monitoring is a solution to this issue that uses aspects to dynamically install the necessary measurement points.

WSML seems an attractive implementation of AOP concept to realize a dyanamic and adaptive environment for web services composition and management. But the web services standards still being in early stages there surely are problems in coming up with the exact notion of semantically equivalent services. Still a solution as presented by JAsCo WSML at least can be taken as a temporary alternative solution to this problem.

6 Conclusion

In this brief survey we looked into two techniques to build adaptive middleware - a purely reflective middleware approach and an AOP based approach. Both these techniques are being taken enthusiastically by the research community as a vehicle for developing middleware platforms to meet the demands of highly dynamic and autonomous computing environments as seen in applications domains like embedded distributed computing and ubiquitous computing.

The obvious difference between a reflective middleware approach (like CompOSE|Q) and the underlying reflective architecture behind AOP middleware (like JAC/JAsCo for instance) is that the former has a fine grained meta-model for core middleware aspects like Qos, Scheduling, Replication etc. Whereas, in AOP middleware the reflective platform is very generic just presenting a meta-model for a particular implementation language like Java, but they provide an expressive language or/and a framework to build software leveraging such reflective platform. Thus, AOP middleware approach seems to provide a reflective skin to cover the composition of distributed aspects like Qos, security, naming etc which are separately built or provided. Consequently the middleware aspects exists as loosely coupled modules provided as standard features along with such a AOP middleware framework.

A more thorough study of these two techniques, especially how they perform and scale in real world situations will help analyze the pros and cons of these approaches, but certainly these techniques have lot in common and the difference merely seems to be at the level they abstract out things or separate the concerns. Also what we have entirely missed in this paper is the ongoing research in middleware community based on software architecture and archtectural styles. Such approach looks into similar problems like adaptation and runtime dynamism but from an architectural viewpoint.

A serious study in a topic as this paper addresses should include references to this alternative approach of exploiting software architecture [DR99] for adaptive middleware. An alternative approach to fill the gap that AOP based solutions have in terms of formal underpinning might be to incorporate a formal architectural style such as RAIC (Redundant Array of Independent Component) [LR02], but that too only addresses one particular problem of putting together equivalent components. At this point it seems AOP based middlware and purely reflective middleware are not completely orthogonal but are complementary.

References

[AWB+93] M. Akşit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object-interactions using composition-filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Processing*, pages 152–184. Springer-Verlag Lecture Notes in Computer Science, 1993.

[Ber94] L. Bergmans. The composition filters object model. Technical report, Dept. of Computer Science, University of Twente, 1994.

[Cou] Geoff Coulson. What is reflective middleware.

[DR99] Elisabetta Di Nitto and David Rosenblum. Exploiting adls to specify architectural styles induced by middleware infrastructures. In *Proceedings of the 21st international conference on Software engineering*, pages 13–22. IEEE Computer Society Press, 1999.

[ea] Nalini Venkatasubramanian et al. Design and implementation of a composable reflective. *The 21st International Conference on Distributed Computing Systems*.

[GKB91] Jim des Rivieres Gregor Kiczales and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991. 335 pages.

[GNV] Sebastian Gutierrez-Nolasco and Nalini Venkatasubramanian. A composable reflective communication framework.

[HCG] Frank Hunleth, Ron Cytron, and Christopher Gill. Building customizable middleware using aspect oriented programming.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming—a critique of pure objects. In *Proc. 1993 Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, September 1993.

[KCBC02] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Commun. ACM*, 45(6):33–38, 2002.

[Kic92] Gregor Kiczales. Towards a new model of abstraction in the engineering of software. *International Workshop on Reflection and Meta-Level Architecture, Tama-City, Tokyo, Japan*, November 1992.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[Lie96] Karl J. Lieberherr. *Adaptive Object-Oriented Software: the Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[LLM99] Karl Lieberherr, David H. Lorenz, and Mira Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA 02115, March 1999.

[LR02] Chang Liu and Debra J. Richardson. Research directions in raic. *SIGSOFT Softw. Eng. Notes*, 27(3):43–46, 2002.

[MO02] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings of OOPSLA '02*, 2002.

[NWP] M. Kircher N. Wang, D. C. Schmidt and K. Parameswaran. Adaptive and reflective middleware for qos-enabled ccm applications.

[PGA01] A. Popovici, T. Gross, and G. Alonso. Dynamic homogenous aop with prose, 2001.

[RCK01] Manuel Román, Roy H. Campbell, and Fabio Kon. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online*, 2(5), 2001.

[RPM] Gerard Florin Fabrice Legond-Aubry Lionel Seinturier Renaud Pawlak, Laurence Duchien and Laurent Martelli. Jac: An aspect-based distributed dynamic framework.

[Sch] Douglas C. Schimdt. R&d advances in middleware for distributed real-time and embedded systems.

[SG] Devon Simmonds and Sudipto Ghosh. Middleware transparency through aspect-oriented programming using aspectj and jini.

[SVJ03] Davy Suvee, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29. ACM Press, 2003.

[VCJ] Bart Verheecke, María A. Cibrán, and Viviane Jonckers. AOP for Dynamic Configuration and Management of Web Services. pages 137–151.

[Ven02] Nalini Venkatasubramanian. Safe 'composability' of middleware services. *Commun. ACM*, 45(6):49–52, 2002.

[VT95] Nalini Venkatasubramanian and Carolyn L. Talcott. Reasoning about meta level activities in open distributed systems. In *Symposium on Principles of Distributed Computing*, pages 144–152, 1995.

[VT01] Nalini Venkatasubramanian and Carolyn L. Talcott. A semantic framework for modeling and reasoning about reflective middleware. *IEEE Distributed Systems Online*, 2(6), 2001.

[WEBa] Web Site: ComPOSE|Q (http://http://www.ics.uci.edu/d̃sm/compose/compose_design.html).

[WEBb] AspectJ official site (http://www.aspectj.org).

[WEBc] Online Documentation for JBoss AOP (http://www.jboss.org) http://www.jboss.org/index.html?module=html&op=userdisplay &id=developers/projects/jboss/aop#framework.

[WEBd] Javassist home page (http://www.csg.is.titech.ac.jp/č̃hiba/javassist/).

[WEBe] The AspectJ Programming Guide (http://www.aspectj.org).

[WEBf] JBoss Java Application Server site (http://www.JBoss.org).

[ZJ03] Charles Zhang and Hans-Arno. Jacobsen. Quantifying aspects in middleware platforms. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 130–139. ACM Press, 2003.

7 Appendix: An outline of ICS 243F term project

As we have seen the potential benefits an AOP based solution like JaSco has for implementing dynamic middleware we intend to look into the possibility of using such a platform for writing a middleware layer to manage web-services. Realizing the constraint we have in terms of time for term project we divide the project into two step goals -

1. We will develop a small bare-bones application that uses free available web-services in the internet and we will put together the application using any one of the standard toolkit to develop a web-services application. We realize that it might be a considerable effort to adopt a new technology and build an application out of it so this is our first priority - to get the application running. During this we believe will be able to get an understanding of at least one approach to composing web services.
2. If we get through our first goal outlined above earlier than we anticipate we will try to investigate how such a loosely coupled application would benefit from using a middleware platform for Web-services, for this we intend to use JAsCo as a dynamic Web Service Management Layer.