Infrastructure for Ubiquitous Computing: Improving Quality with Modularisation

Jennifer Munnelly Siobhán Clarke

Distributed Systems Group
Trinity College Dublin
munnelj, sclarke @cs.tcd.ie

Abstract

Infrastructural software encompasses a host of technologies that are required by ubiquitous computing applications in their environment. This includes the provision of fundamental communication mechanisms, resource management, network management and systems software. The functionality provided by infrastructural software inherently crosscuts the applications they support resulting in poorly modularised code, negatively affecting the quality of the software. Aspect-oriented software development (AOSD) modularisation capabilities attempt to improve software quality. To assess any variations in quality, software quality factors must be defined and measured. We identify a set of infrastructural concerns in the domain of ubiquitous computing and outline an aspect-oriented design to improve software modularity. We define a set of software quality factors and, using the Goal-Question-Metric (GQM) approach, a method for their quantifiable measurement. We describe a proposed case study for a comparative study of quality in ubiquitous computing infrastructural software.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics

General Terms Software Quality, Metrics, Measurement

Keywords Quality, Metrics, Infrastructural Software, AOP, Modularity

1. Introduction

There are two main perspectives in this paper; the first is the identification and modularisation of concerns in ubiquitous computing infrastructural software to improve the quality of the software. The second is a perspective on how the improvement in quality can be quantifiably measured.

Infrastructural software encompasses a host of technologies that are required by applications in their environment. Below the application layer, infrastructural software supports the technical requirements for all applications. This entails the provision of fundamental communication mechanisms, resource management, network management and systems software. Section 2 discusses the variations in infrastructural software according to the environment in which the applications run.

Infrastructure is used by multiple parts of applications, causing intertwined functionality or crosscutting. Poorly modularised code reduces quality in software. Aspect-oriented approaches attempt to increase software quality by improving modularity. We adopt this approach in the attempt to improve quality in the infrastructural software used by ubiquitous computing applications. Commonalities in infrastructural software can be abstracted to identify concerns within the domain. As part of the development of the UILE framework [1], we identify a set of concerns common to infrastructural software in ubiquitous computing, in particular software used by mobile and context-aware applications. We then propose an aspect-oriented design for the modularisation of these concerns and describe two concern designs in detail.

The proposed modeling of infrastructural concerns can improve the quality of ubiquitous computing applications by increasing the modularity of the software. To assess the proposed modularisation, quantification of the software quality improvements are required. Software quality reflects how well software has been designed and implemented. Software engineering terms known as *ilities* [2] are indicators of software quality. We define a set of these ilities as software quality factors. They are comprehensibility, manageability, maintainability, scalability, testability, reusability, and usability. By measuring these factors we can calculate and reason about any improvement or deterioration in quality. This in turn allows us to reason about the cause of the change in quality and to assess its effectiveness using empirical evidence. Using the GQM [3] approach the high-level software quality factors can be mapped to low-level code based metrics, enabling the quantifiable measurement of quality.

The rest of the paper is structured as follows; section 2 describes infrastructural software, section 3 explains the aspect-oriented approach to modularising identified ubiquitous computing infrastructural concerns, section 4 defines a set of software quality factors and how they are measured using the GQM approach. In section 5 we describe the ongoing case study of a comparative evaluation of the aspect-oriented design and an object-oriented design using the approach and technique described in this paper. Section 6 discusses related work.

2. Infrastructural Software

Infrastructural software includes the set of technologies which are required by applications to execute in their environment. The software below the application supports, manages and provides resources to the application using the infrastructure.

Infrastructural software can take on different definitions according to the context within which it is referred to. For example, basic

operating systems software is infrastructural and common to all applications but in a distributed context the infrastructure expands to include middleware.

The UILE framework aims to support the development of ubiquitous computing applications. We therefore examine infrastructural software in the domain of ubiquitous computing, specifically mobile and context-aware applications. For the purpose of this paper we view infrastructural software as the underlying concerns related to the technical infrastructure in which the ubiquitous computing application runs, e.g., available computation resources, communication capabilities, network configuration and information pertaining to the system components and their configuration [4].

2.1 Ubiquitous Computing Infrastructural Concerns

From previous work in the area of mobility [5] and context-awareness [4], we examined the commonalities in ubiquitous computing applications. From the analysis we have identified a set of infrastructural concerns that crosscut applications. The set of concerns that arose as infrastructural software concerns are:

- Roaming
- Discovery
- Ad-Hoc Networking
- Limited connectivity
- Quality of service
- System Context Adaptation

Roaming refers to the movement of a user involving changes in network or devices. Discovery handles the location and knowledge management of other nodes and services. Ad hoc networking addresses the routing of data between nodes in an ad hoc environment. Limited connectivity deals with the unreliability of network connections in distributed mobile environments and the contingency plans required on disconnection. Quality of service refers to the capabilities and resources associated with particular networks and the management of application quality of service requirements. Finally, adaptation may take place in response to changes in system context i.e., performance, hardware availability and network conditions [4].

3. Modularisation of Infrastructural Concerns

Infrastructure underlies all applications and is used by multiple parts of the application, causing inherent crosscutting. This leads to badly modularised code reducing the overall quality of the software. Therefore aspect-oriented approaches are particularly suitable for the modularisation of these concerns. We propose an aspect-oriented framework for these concerns in order to increase modularity resulting in better quality software. We have investigated the modularisation of these concerns [5] and for the purpose of this paper we describe the aspect-oriented design for three of these concerns in detail. We use the Theme/UML [6] aspect-oriented design approach to illustrate the modularisations. Firstly we describe the limited connectivity concern and how it can modularise the functionality required in the event of disconnection from a network. Secondly we describe a concrete example of the discovery concern modularising the service discovery process. Finally we describe the modularisation of system context adaptation.

3.1 Limited Connectivity

Ubiquitous computing applications migrate between periods of direct connectivity, intermittent connectivity, and disconnectivity and can have varying amounts of bandwidth available to them as well as different network latencies depending on how they connect to the distributed application servers. Disconnections may occur due to poor network coverage, failed network handovers, severe network congestion and client hardware failures. An application should handle abrupt disconnections by adequately detecting these disconnections in a timely fashion, and provides facilities for rollbacks to ensure consistency in system state.

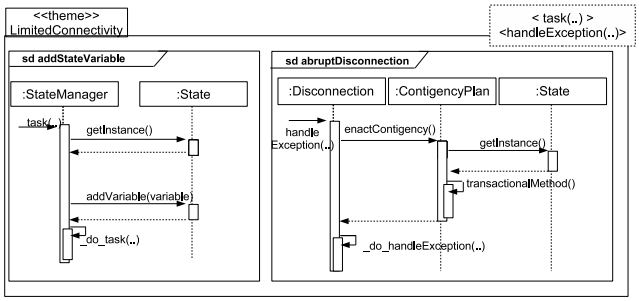

Figure 1. Limited Connectivity Module

Modularisation of Limited Connectivity The Limited Connectivity module is responsible for ensuring that the application is in a consistent state when there is no connection to the network. It serves two purposes. First, it is responsible for logging any state changed by the application. By modularising this functionality, the various logging mechanisms can be integrated for different purposes, e.g. coarse grained logging or more sophisticated fine-grained logging techniques. Second, a contingency plan is applied in the case of abrupt disconnections to yield a consistent state. This contingency can be implemented as a transactional service which rolls back any actions of the application which might be not fully executed. More sophisticated contingency plans might cache data in order for the application to continue even without network access [7].

Figure 1 illustrates the Limited Connectivity module. When an operation occurs that changes the state of the application, a persistent record of the state should be updated. In the event of network failure or disconnectivity, a contingency plan should then be applied, making use of the application state that has been recorded.

We define the Limited Connectivity module with four entities:

The *ContingencyPlan* class acts as a placeholder for a concrete contingency plan implementation that should incorporate the various persistence concerns, such as transaction management, which are necessary for handling abrupt disconnections and rectifying the inconsistent program state that follows.

The *State* class is responsible for storing state information about the user/client application. Upon the creation of a concrete contingency plan the State class should be subclassed with methods capable of recording the desired data.

The *StateManager* class is responsible for the logging of the current state of the application.

The *Disconnection* class is responsible for invoking the implemented contingency plan whenever a network failure is detected.

The sequence diagram in Figure 1 depicts the augmented application behaviour using the Limited Connectivity module. Any method that alters the state of the user or client application is intercepted, so that the State object can be updated with the altered state information in case a network failure occurs after the alteration. The second scenario is depicted in Figure 1 and applies whenever there is an abrupt disconnectivity raised in the application due to network exception. The execution of a Contingency plan would then ensure that the application is able to continue after the loss of network connection, e.g. by rolling back previous updates or caching data the application is likely to use.

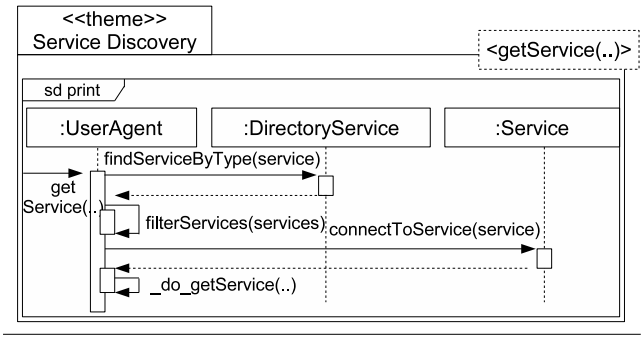

Figure 2. Service Discovery Module

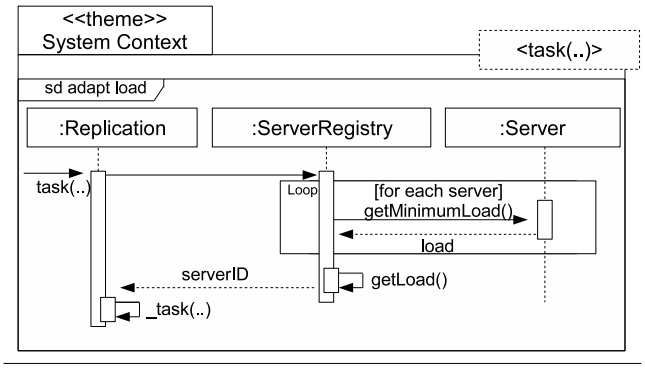

Figure 3. System Context Adaptation Module

3.2 Service Discovery

Applications require the ability to dynamically discover and access necessary services. To facilitate dynamic service discovery, services advertise their existence on the network. Clients request a particular service and obtain information on how to access the most suitable service in their current context.

When a service is accessed, the application has to ensure that the service is still reachable, and that the connection fulfills the specific requirements of the service, e.g. network bandwidth. However, these requirement checks are usually wrapped around each call of a service in the base code.

Modularisation of Service Discovery The Service Discovery module provides means for clients to dynamically discover and access necessary services. For services, it provides mechanisms to advertise their existence, and to update their information in directories. When a mobile user requires access to a service, it uses the Service Discovery module as illustrated by the *getService* template in Figure 2, to acquire information of services available and how to access these services.

In the Service Discovery module we define consists of three components.

The *DirectoryService* stores information about active services accessible through its supported network, as well as providing interfaces to allow services to maintain their entries in the directory agent.

The *Service* maintains the state of a service. It listens for requests and sends an advertisement back to the requester. If the client wishes to use the service, the service agent must be able to accept a connection on behalf of the service. This can be implemented by concrete service types e.g., print service.

The *UserAgent* provides an interface to allow client applications to search for desired services, either through the use of a directory agent, or directly by multi casting to the network. As potentially many replies could come back in response to the request, the user agent should be able to select the most suitable service based on the application's current context, before connecting to it.

The sequence diagram in Figure 2 depicts a use of the Service Discovery module. The *User Agent* intercepts any calls to a service. Then it searches for potential services or resources related to that service on the network by using the *Directory Service*. The *User Agent* then filters the resulting services to select a service that is the most suitable to the application's current context.

3.3 System Context Adaptation

Adaptation in context-aware applications takes place in response to changes in the application's environment. System context refers to all context related to the (technical) infrastructure, in which an application runs. It includes all available system components and information pertaining to the components and their configuration [8].

System context is usually obtained explicitly, e.g., using deployment scripts. Existing component architectures, like Enterprise Java Beans (EJB) [1] or Common Object Request Broker Architecture (CORBA) [9], alleviate the development of distributed applications by providing services like persistence, naming and concurrency handling. However, the adaptation of an application to changing system conditions, e.g., load adaptation, cannot be modularised cleanly and usually exhibits crosscutting behaviour.

Modularisation of System Context Adaptation The system context module provides the monitoring of system context and enables applications to run in different configurations, based on performance requirements, hardware availability and network conditions. Our approach is taken from Atlas [10], a web-based application, which uses aspects to support different architectural configurations.

The system context module is illustrated by the *task* template in Figure 3. A *ConfigurationContext* aspect generalises the different configurations an application might be running in and is realised as an abstract aspect. Each aspect that inherits from *ConfigurationContext* contains the code particular for a specific configuration of the application. For instance, the default configuration could be a single server aspect, *Server*, that handles all client requests for an application. A *Replication* aspect allows the execution of the application on various nodes and is responsible for the dynamic dispatching of client calls to the server with the least workload. The aspect has a reference to a *ServerRegistry* class, which keeps track of all available servers in the system. A server is represented by a *Server* class and encapsulates all capabilities of a node in the system. *Replication* intercepts client calls for the application. It then forwards these calls to the server with the current minimum workload by referring to the information obtained from the *ServerRegistry*.

Few context frameworks deal with system context as first-class context. The Wildcat framework [11] provides a context domain for system resources, e.g., memory, discs, devices. This domain can be queried by the application, either in a synchronous way or by listening to events when the status of one of the attributes changes. However, dealing with adapting to the changed system context is part of the base code and therefore is not encapsulated cleanly. Using the described AOP approach, UILE cleanly encapsulates context adaptation with positive effects i.e., more modular code [4].

4. Evaluation of Software Quality

This section outlines how to assess the improvement in software quality that is realised by implementing the identified concerns

[1] http://java.sun.com/products/ejb/

using the proposed modularisation. Software quality is a measure of the correct and efficient design and implementation of software both as a whole and as individual modules. Good modularisation improves the overall quality of software, but this claim should be verified by empirical evidence. Quality software exhibits well known indicators that signify the strengths of the software. These indicators are known as the *ilities* of software engineering and are established software quality factors [2]. From the vast quantity of research in software engineering, we define a concise set of ilities that together depict a comprehensive view of quality in software. These are:

- Comprehensibility
- Maintainability
- Manageability
- Scalability
- Testability
- Reusability
- Usability

We describe these software quality factors later in this section listing idiomatic terms by which they are also known. Higher-level quality concepts may be considered using the given set of quality factors e.g., modularity. Parnas [12] outlines the expected advantages of modular programming as better maintainability, manageability and comprehensibility. We can therefore use these advantages as indicators to identify modularisation improvements [4].

4.1 Measuring Software Quality Factors

To assess the overall quality of a piece of software a quantifiable measurement must be calculated for the software quality factors. Although metrics exist for low-level code based concepts, higher-level software quality factors such as modularity and evolvability are more difficult to quantify. For these high-level factors we employ the GQM approach to map low-level code based metrics to the high-level software quality factors. This enables quantifiable measurement which realises the ability to compare and contrast quality in infrastructural software systems and components.

4.2 Goal-Question-Metric Approach

We use the GQM approach enable low-level metric results to be reasoned about and traced to quantifiably measure software quality factors. The approach uses a three tier architecture made up of conceptual level goals, operational level questions and quantitative level metrics. High-level concepts are often the goals of measurement. In this case a set of software quality factors are our conceptual level goals. Operational level questions are functional representations of the goals above them. These questions are then answered by quantifiable level goals which yield tangible results allowing the eventual measurement of the initial goals. Figure 3 illustrates the high-level quality factors as conceptual level goals which are mapped to operational level questions which are in turn answered by quantifiable level metrics. The following section describes each software quality factor briefly including idiomatic terms by which it may also be known and explains how the GQM approach is applied.

4.2.1 Comprehensibility

Comprehensibility is the ease with which application developers understand the software. It is also referred to as understandability and is a key indicator of good software quality with benefits such as reduced development time and increased productivity. The goal of comprehensibility is effected by the complexity of a module.

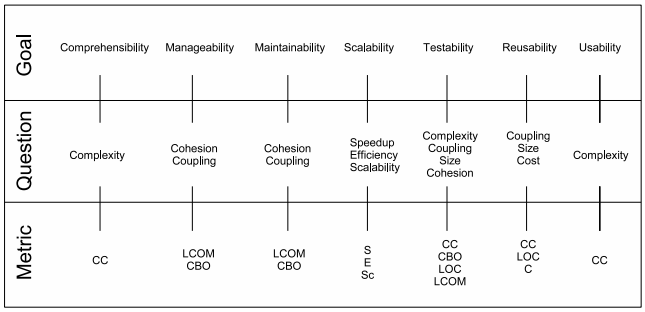

Figure 4. Goal-Question-Metric Approach

Complexity is therefore the operational level question representing comprehensibility as the effort to understand a module is greatly increased by high complexity in a module. The well known cyclomatic complexity [13] metric is the quantitative level measurement used to indicate complexity.

4.2.2 Manageability

Manageability is the ease with which software components can be developed in isolation and later composed. It is also referred to as composability. Manageability is measured by metrics that can identify the independence of a module, enabling the module to be developed and modified in isolation e.g., coupling and cohesion [4]. The goal of manageability is therefore represented by the questions of coupling and cohesion. These software attributes have established metrics (e.g, lack of cohesion between modules, coupling between objects, efferent coupling, afferent coupling) associated with them that yield qualitative level results.

4.2.3 Maintainability

Maintainability is the ease with which software components can be changed and updated with least disruption to the software system. It is also known as evolvability, extensibility, adaptability, changeability, customisability, and flexibility. The maintainability goal is effected by the links a module has with others, i.e., fewer dependencies make changes easier and less disruptive. Again, the operational level questions coupling and cohesion enable the goal to be assessed using their corresponding quantitative level metrics.

4.2.4 Scalability

Scalability is the ease with which a system can be extended to a larger magnitude. Scalability is a crucial software quality factor in infrastructural software due to it being the fundamental basis for all applications. It must scale both efficiently and correctly. The goal of scalability can be measured using three operational level questions, each with their own metrics: speedup metrics, efficiency metrics, and scalability metrics [14].

4.2.5 Testability

Testability is the ease with which checks can be carried out on various aspects of the software. The goal of testability can also be examined by assessing the dependencies within the software as fewer dependencies increase its ability to be tested [15]. At an operational level, complexity, coupling, size and cohesion all effect the goal of testability. Established metrics including cyclomatic complexity, coupling between objects, lines of code and lack of cohesion in modules can all be used to measure testability an a quantitative level [15].

4.2.6 Reusability

Reusability is the extent to which software components can be used again in other systems. Reusability as a goal decreases developer time, effort and cost and is a desirable quality attribute. To be reusable, a module must be well encapsulated with as few dependencies as possible. The Chidamber and Kemerer metrics suite [19] links coupling with the goal of reuse making it an operation level question. Two other questions are used in the estimation of the reuse of a software module: size and cost. Metrics based on the computation of size and cost measurements include additional development cost, new or changed source instructions, relative cost of reuse, reuse cost avoidance and source instructions reused by others [20].

4.2.7 Usability

Usability is the ease with which application developers learn and use the software. This goal is directly effected by how difficult the software is to understand for the application developer. Using the operation level question of complexity allows the application of complexity metrics to quantifiably measure how easily the software may be used.

5. Current Work

The proposed modularisations for ubiquitous computing infrastructural concerns are design elements of the UILE framework. These concerns, along with others in the areas of mobility and context-awareness are currently being designed, implemented and evaluated as part of the framework development.

To quantify the improvements in quality realised by the proposed infrastructural software modularisations we use a case study for evaluation purposes. This case study is ongoing and is based on a comparative study using an existing object-oriented implementation of a ubiquitous computing application incorporating mobility and context awareness. The application is being refactored to implement the aspect-oriented modular designs to enable the encapsulation of crosscutting concerns, specifically the infrastructural concerns outlined in section 2.1. Both applications can then be analysed using the GQM approach described in section 4.2. Appropriate object-oriented metric tools [2] and aspect-oriented extensions [3] compute metrics based on both applications. The quantitative results obtained can then be used to answer the operational level questions e.g., high or low coupling, very complex code or code with low complexity. These can then be used to determine the degree to which each software quality goal is achieved. Modularity levels can be compared by inspection of the software quality factors indicating good modularity. An overall quality comparison can be conducted using the acquired results by comparison analysis of the software quality factors.

6. Related Work

The use of AOSD in modularising infrastructural software is dispersed into the various areas that combine to form infrastructure used by applications. AOSD has been used successfully to modularise distribution [21] in applications including JMS and RMI. Embedded systems software has also benefited from an aspect-oriented language targeted at development in the area [22]. Improvements in quality [23] using AOSD have also been noted in areas including garbage collection [24], context adaptation [4] and transaction management [25].

Systems software improvements using AOSD are also common. Operating systems can be customised using aspect-oriented frameworks to show relationships between features more clearly[26]. Operating systems themselves can be designed using AOSD enabling improved customisability [27].

Qualitative studies on modularity effects in infrastructural software are less common. Operating system code has been shown to improve in quality using AOSD and this is supported by quantitative analysis [28] [29].

There is a huge body of work in the area of AOSD in middleware proving the beneficial effects achievable by its modularisation capabilities. Areas where AOSD have made notable improvements are customisable middleware e.g.,[30] [31] and middleware itself.[4]

Quantifiable studies substantiate the improvements in quality using AOSD in middleware platforms [32] and improvements in modularity [31]. Scalability is also empirically shown to be improved by AOSD in middleware [33].

Acknowledgments

This work is funded by Science Foundation Ireland under the Research Frontiers Program. The authors would like to thank Serena Fritsch, Andrew Jackson and Neil Hatton for previous work.

References

[1] Fritsch, S., Munnelly, J. and Clarke, S. 2006. Towards a Domain-Specific AOP language for Ubiquitous Computing. In Proceedings of the AOSD Workshop on Open and Dynamic Aspect Languages (ODAL 2006), March 2006, Bonn, Germany.

[2] F. Manola, Providing Systemic Properties (Ilities) and Quality of Service in Component-Based Systems, Object Services and Consulting, Inc., Technical Report 1999.

[3] V. R. Basili, G. Caldiera and H. D. Rombach: Goal Question Metric Paradigm. Encyclopedia of Software Engineering, Volume 1, pp. 528-532, (1994).

[4] Munnelly, J., Fritsch, S., and Clarke, S. 2007. An Aspect-Oriented Approach to the Modularisation of Context. In Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications (March 19 - 23, 2007). PERCOM. IEEE Computer Society, Washington, DC, 114-124.

[5] AOSD-Europe Network of Excellence: A domain analysis of key concerns: known and new candidates (2006) http://www.aosd-europe.net/deliverables/d43.pdf.

[6] S. Clarke and E. Baniassad. Aspect-Oriented Analysis and Design. The Theme Approach. Addison-Wesley, Boston, USA, 2005.

[7] Kuenning, G. H. and Popek, G. J. 1997. Automated hoarding for mobile computers. In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (Saint Malo, France, October 05 - 08, 1997). W. M. Waite, Ed. SOSP '97. ACM, New York, NY, 264-275.

[8] Schmidt, A., Beigl, B. and Gellersen H-W. There is more to Context than Location. Computers and Graphics Journal, Elsevier, Volume 23, No.6, pp 893–902, 1999.

[9] Object Management Group, The Common Object Request Broker: Architecture and Specification, 2nd ed., July 1995.

[10] Kersten, M. A. and Murphy, G. C. 1999 Atlas: a Case Study in Building a Web-Based Learning Environment Using Aspect-Oriented Programming. Technical Report. UMI Order Number: TR-99-04., University of British Columbia.

[11] David, P. and Ledoux, T. 2005. WildCAT: a generic framework for context-aware applications. In Proceedings of the 3rd international Workshop on Middleware For Pervasive and Ad-Hoc Computing (Grenoble, France, November 28 - December 02, 2005). MPAC '05, vol. 115. ACM, New York, NY, 1-7.

[12] Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. Commun. ACM 15, 12 (Dec. 1972), 1053-1058.

[2] e.g., Cyvis, http://cyvis.sourceforge.net/index.html

[3] e.g., Aop Metric Suite, http://aopmetrics.tigris.org/metrics.htm

[4] JBoss http://labs.jboss.com/jbossaop/

[13] McCabe, T.J.: A complexity measure. In: ICSE '76: Proceedings of the 2nd international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1976) 407

[14] Jogalekar, P. and Woodside, M. 2000. Evaluating the Scalability of Distributed Systems. IEEE Trans. Parallel Distrib. Syst. 11, 6 (Jun. 2000), 589-603.

[15] Bruntink, M. and Deursen, A. v. 2004. Predicting Class Testability using Object-Oriented Metrics. In Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE international Workshop on (Scam'04) - Volume 00 (September 15 - 16, 2004). SCAM. IEEE Computer Society, Washington, DC, 136-145.

[16] Ciraci, S. and van den Broek, P.M. (2006) Evolvability as a Quality Attribute of Software Architectures. In: The International ERCIM Workshop on Software Evolution 2006, 6-7 Apr 2006, LIFL et l'INRIA, Universite des Sciences et Technologies de Lille, France. pp. 29-31.

[17] Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M. 1997. Metrics and Laws of Software Evolution - The Nineties View. In Proceedings of the 4th international Symposium on Software Metrics (November 05 - 07, 1997). METRICS. IEEE Computer Society, Washington, DC, 20.

[18] L. C. Briand and J. Wuest, "Empirical studies of quality models in object-oriented systems," Advances in Computers, vol. 59, pp. 97-166, 2002.

[19] Chidamber, S. R. and Kemerer, C. F. 1994. A Metrics Suite for Object Oriented Design. IEEE Trans. Softw. Eng. 20, 6 (Jun. 1994), 476-493.

[20] Poulin, J. S. 1996 Measuring Software Reuse: Principles, Practices, and Economic Models. Addison-Wesley Longman Publishing Co., Inc.

[21] Soule, P., Carnduff, T., and Lewis, S. 2007. A distribution definition language for the automated distribution of Java objects. In Proceedings of the 2nd Workshop on Domain Specific Aspect Languages (Vancouver, British Columbia, Canada, March 12 - 12, 2007). DSAL '07. ACM, New York, NY, 2.

[22] Sousan, W., Winter, V., Zand, M., and Siy, H. 2007. ERTSAL: a prototype of a domain-specific aspect language for analysis of embedded real-time systems. In Proceedings of the 2nd Workshop on Domain Specific Aspect Languages (Vancouver, British Columbia, Canada, March 12 - 12, 2007). DSAL '07. ACM, New York, NY, 1.

[23] Laddad, R. and Alexander, R. 2003. Aspect-oriented programming will improve quality, Quality Time. IEEE Softw. 20, 6 (Nov. 2003), 90-93.

[24] Gibbs, C. and Coady, Y., OASIS: Organic Aspects for System Infrastructure Software Easing Evolution and Adaptation through Natural Decomposition, RAM-SE'04-ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution, Proceedings, Oslo, June 15, 2004.

[25] Fabry, J. and D'Hondt, T. 2006. KALA: Kernel Aspect language for advanced transactions. In Proceedings of the 2006 ACM Symposium on Applied Computing (Dijon, France, April 23 - 27, 2006). SAC '06. ACM, New York, NY, 1615-1620.

[26] Park, J. and Hong, S. 2003. Customizing Real-Time Operating Systems with Aspect-Oriented Programming Framework, In Proceedings ofSoC Design Conference, 2003 pp. 966-970

[27] Lohmann, D., Scheler, F., Tartler, R., Spinczyk, O., and Schrder-Preikschat, W. 2006. A quantitative analysis of aspects in the eCos kernel. SIGOPS Oper. Syst. Rev. 40, 4 (Oct. 2006), 191-204.

[28] Coady, Y. and Kiczales, G. 2002 Back to the Future: a Retroactive Study of Aspect Evolution in Operating System Code. Technical Report. UMI Order Number: TR-2002-11., University of British Columbia.

[29] Greenwood, P., Bartolomei, T., Figueiredo, E., Dosea, M., Garcia, A., Cacho, N. et al. 2007. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In Proceedings of the 21th European Conference on Object-Oriented Programming (ECOOP'07). Berlin , July 2007.

[30] Pratap, R.M.; Hunleth, F.; Cytron, R.K., 2004. Building fully customisable middleware using an aspect-oriented approach, Software, IEE Proceedings - , vol.151, no.4, pp. 199-216, 6 Aug. 2004

[31] Cacho, N., Batista, T., Garcia, A., Sant'Anna, C., and Blair, G. 2006. Improving modularity of reflective middleware with aspect-oriented programming. In Proceedings of the 6th international Workshop on Software Engineering and Middleware (Portland, Oregon, November 10 - 10, 2006). SEM '06. ACM, New York, NY, 31-38.

[32] Zhang, C. and Jacobsen, H. 2003. Quantifying aspects in middleware platforms. In Proceedings of the 2nd international Conference on Aspect-Oriented Software Development (Boston, Massachusetts, March 17 - 21, 2003). AOSD '03. ACM, New York, NY, 130-139.

[33] Colyer, A. and Clement, A. 2004. Large-scale AOSD for middleware. In Proceedings of the 3rd international Conference on Aspect-Oriented Software Development (Lancaster, UK, March 22 - 24, 2004). AOSD '04. ACM, New York, NY, 56-65.