Aspects Pattern Oriented Architecture For Distributed Adaptive Mobile Applications

Dimitar Birov

Abstract: *Adaptive applications behave differently, according to changes on the environment. Aspect oriented programming (AOP) propose approach of implementation of adaptability of a software through special form of concern composition. In this paper we discuss how aspect-oriented approach facilitates the design of distributed adaptive wireless applications and propose integrated software architecture framework based on known (aspect) design patterns architecture building approach. proposed framework allows to achieve appropriate level of middleware transparency, consistency, flexibility, maintainability, and modifiability of application.*

Key words: *Aspects, Aspect-Oriented Programming, Design Patterns, Middleware, Software Architecture, Wireless Application*

INTRODUCTION

To address the constraints imposed by mobile systems (and to take advantage of the opportunities provided by these systems), it is necessary to abandon the current paradigm [3] in software design where software capabilities are determined at build time. Aspect-oriented programming (AOP) [10, 14] offers an alternative paradigm for software development. Some middleware environments demand flexible software architecture and functionality became a dynamic property and application can be adapted to the environment at run-time.

Adaptive application means capability of application to modify its own behavior in response of changes in its operating environment [21]. Adaptive applications behave differently, according to changes on the environment. Implementing this kind of application involves complex issues, so it is important to provide adaptive behaviour following quality and productivity factors.

Aspect oriented programming (AOP) [10,14,16] propose approach of implementation of adaptability of a software through special form of concern composition [2, 25].

In this paper we discuss how aspect-oriented approach (AOP) facilitates the design of distributed adaptive wireless applications. We propose integrated software architecture framework based on known (aspect) design patterns architecture building approach, implemented with programming languages family tools such like Java, AspectJ, J2ME, J2EE, Jini. This framework allows to achieve appropriate level of middleware transparency, consistency, flexibility, maintainability, and modifiability of application.

Separation of concerns was recognized as a fundamental mechanism for managing complexity of software systems. Software Engineering recognizes *functional* and *nonfunctional requirements* to the software. Object oriented paradigm and design technology capture well functional requirements and *core concerns*. Aspect Oriented design and programming propose way for implementing nonfunctional or *system-level concerns*.

Different parts of wireless distributed adaptive applications are discussed, developed and designed in the software development community. In their AspectJ evaluation paper Dantas and Borba [8] proposed closed-adaptive behaviour client side adaptive application – it is not able to support the addition of new behaviour during runtime. Adaptive behaviour should be programmed before deployment, but activated or deactivated in response to environment changes. This constraint is imposed by compile time weaving mechanism of ApectJ and missing of loading code at runtime in J2ME. They identify some adaptive concerns such like Customization, Screen, Internationalization which are included as a part of proposed architecture in this paper.

Design patterns, aspects and components are three orthogonal concepts [4] which arised from a common source – idea to modularize software and deal effectively with software complexity. In despite of that they are developed independently during last decade, consequences of their interactions between three of them is not well studied and hide a great potential towards the decision of software crisis problem. This issue is an attempt to add to this subject.

AOP allows integration of the specific concerns – access control, distribution and synchronization [18], real-time concerns [1] – into an existing application even if it affects many decomposition units of the original application. From self-organization perspective, the *binding-time* of the concern is crucial. For instance J2ME don't allow building time weaving. AOP can be implemented using compile-time [15, 24], load-time [7, 13] or run-time [5, 17] binding techniques.

This report is organized as follow: next section introduces the aspect-oriented concepts. Sections after that presents distributed aspect patterns, adaptability aspect patterns, and some natural emerged aspect patterns integrated in common pattern software architecture. Aspect implementations tied all them up in a consistent architecture. Last section provides conclusions.

ASPECT ORIENTED PROGRAMMING

After Kiczales [14] clarified the meaning of AOP and gave a brief outline of its major tenets, most of the current literature focuses on the challenges of crosscutting concerns or the separation of concerns. Crosscutting concerns are elements of software, which cannot be expressed, in any functional unit of the programming language's abstractions. In object-oriented programming parlance, crosscutting concerns are elements of an application which cannot be cleanly captured in a method or class and so has to be scattered across many classes and methods. Such concerns include: applying design patterns, applying synchronization policies, applying exception handling, error-checking or fault tolerance concerns, sharing resources, security issues, performance measures, etc.

Aspect-oriented programming allows us to decompose software systems in different dimensions. We can use a vertical decomposition process to establish the primary decomposition model of the architecture. We then use aspect-oriented techniques to "horizontally" compose or to "superimpose" the implementation for orthogonal design requirements onto the primary model, without modifying the existing architecture. We refer to that decomposition process as the horizontal decomposition [28].

Aspect oriented solution consists of three parts: the *application* holds the application or business code (called "core code"), the *aspect part*, where the aspect code resides and a coordinating component, and weaving mechanism, provided by the AOP system. Crosscutting concerns modularized as aspects are used to design (horizontal) decomposition and are provided by aspect oriented language such as AspectJ.

In addition to conventional Java language features, AspectJ defines a set of new language constructs to model the aspects. A *joinpoint* represents an interception point in the execution flow of the component program. For convenience and elegance, a *pointcut* construct can be used to denote a collection of joinpoints. Actions can be triggered before, after, or in place of the program execution when a joinpoint is reached. These actions are defined using aspect language specific constructs called *advices*. An *aspect* module in AspectJ contains pointcuts and the associated advices. It also contains intertype declarations, which are reused to declare new members (fields, methods, and constructors) in other types.

The ability to switch aspects on and off during runtime [20] allows to change the aspect context of an application during runtime and every context change can alter the code of "aspectized" methods. Every time a method of the object is executed this runtime type is used to decide which aspectual code is executed in addition to/instead of the original method.

DISTRIBUTED ASPECT PATTERN

Middleware is an enabling layer of software that resides between the application and the networked layer of heterogeneous platforms and protocols. It decouples applications from any dependencies on the underlying layers, which consist of heterogeneous operating systems, hardware platforms and communication protocols. Middleware is a key component in integrating highly distributed, mobile, computing resources. Middleware layer decouple the deployment of services from the underlying network infrastructure. This middleware layer should provide interfaces to application service providers (API).

Implementation of a distributed application requires distribution of modular components among locations [26]. Conventional Object-Oriented Programming (OOP), doesn't offer easy way of changing the design decision about distribution, because of tangled code (code with different concerns interlacing to each other) and scattered code (code regarding one concern spread in several units of the system). AOP offers an appealing approach since it allows the design decisions regarding different distributions policies to be specified separately, making it easy to design them and to switch from one to another [6].

The natural appeal of AOP to distributed applications stems from the fact that flexibility, easy of use, and usability of distributed applications engender the evolution of many middleware technologies. Implementation of distributed system requires taking into account following requirements. A different middleware proposing distributive environment exist – RMI, Jini, DCOM, CORBA. The application can use different middleware at the same time as well the application should be completely independent of the communication API. This facilitates system maintenance and separate communication code from business code and user interface code. A changing the communication API without impacting other system code allows two clients accessing the system, one using RMI and the other Jini for example. Middleware transparency (abstractions that capture those elements of the application specific to the middleware and allow seamless integration of the abstracted elements into an application) property proposes changing of middleware without changing or recompiling applications source code.

Distributed Aspect Pattern defines distribution concerns using aspect for crosscutting client side (client aspect), server side (server side aspect). Server side aspect crosscut server component preparing it to respond of remote call. Server side component can implements Façade and Singleton design patterns [11]. Server side aspect should intercept initialization of the server component in distributive environment and wrap around remote call methods using Wrapper-Façade pattern [22] for example. Client side aspects are responsible for redirecting calls through remote calls to the server through a fixed set of functions which set internal code to handle discovery events and services, interacting with the lookup service, search for services, maintaining service leases.

Exception handling aspect is common concern (service) which crosscutted all classes and components of the application. Proposed by Soares and Borba Pattern of Distribution aspects [23] integrate exception handling concern as a part of the pattern. We cut this aspect from the pattern and expand exception handling concern as a common aspect for all participants in the framework.

ADAPTABILITY ASPECT PATTERN

Adaptability has become strong requirement [19] to middleware and this way to mobile and wireless applications. Adaptive behaviour implementation needs a clear concern about adaptability, which will affect final application to responds appropriately to context changes. Contextual information obtained from new sources may change and lead to different applications behaviour. Adaptive distributed application design requires separating adaptability concerns from other concerns, which will improve reuse and maintainability. Using aspects makes the application adaptive in a modularised and non invasive way.

Adaptability of the system requires differentiating between client set of aspects and server side client aspects. Additionally to this we have a common set of aspects common to client and server.

Changing contextual information should not cause a significant impact on the system as well the adaptability functionality might be either plugged in/out and also turned on/off, because the ways to access the context and the behaviours triggered by environment state may change a lot.

Wireless and mobile terminals add additional requirement – the application can be implemented in any platform, from embedded devices, such as cellular phones, to enterprise applications.

In order to support adaptability we need aspects, which are able to crosscut some application execution points and change their normal execution. A module for monitoring the context changes they interact with will be responsible for support context change data to the core functionality units. A group of auxiliary classes collect adaptation functionality and perform the application changes. Auxiliary classes are used to avoid requiring all developers to know an AOP language and to avoid code tangling in the adaptive behavior implementation. The auxiliary classes interact with a module responsible for obtaining dynamic data specifying how the application should adapt in a specific situation and this way we increase level of flexibility.

The Adaptability Aspects architectural design pattern [9] identifies five basic participants. *Core application* enclose main application functionality – business processes, GUI, etc. *Adaptability aspects* implement the adaptability concern and control and modify the behaviour of the base application and specify how the functionalities should be changed to adapt to contextual changes. The different (adaptation) tasks could be delegated to the auxiliary classes. In order to provide adaptive behaviour aspects use auxiliary classes. This approach improves reuse and don't requires from the developers of this module to know an AOP language and to develop auxiliary classes independently. System architect specify just an interfaces and functionality of these classes. The aspects invoke methods of auxiliary classes. Auxiliary classes communicate with the *adaptation data provider* module, which consists of classes responsible for providing data for dynamic adaptations according to context changes, in order to obtain dynamic data for the adaptation. The same context change can lead to different behaviours in different moments according to the data provided by this module. These classes can be organized as an Adaptive Object-Model (AOM) [27]. *Context manager* module is responsible for monitoring and analyzing context changes and triggering adaptive actions implemented by the aspects as well aspects can call it in order to obtain information about the context. Its implementation can be based on a variation of the Observer pattern [11], or on its implementation with aspects [12]. In this way, new mechanisms for accessing the context can be easily supported without significant impact on the application.

NATURALLY EMERGED ASPECTS

Many exceptions of all kinds are thrown in a distributed application, variety of forms of interruptions and failures could occur also. An aspect to trap all uncaught exception is very useful. A service monitor is aspect that monitors interactions between clients and servers. Each client and each server may choose its personal monitor.

Concerns like fault-tolerance, caching, object transmission on demand can be included and integrated to the framework using appropriate aspects. This will increase system robustness and efficiency.

CONCLUSIONS

Integrated pattern framework architecture for distributed adaptive wireless applications have been presented. The architecture increase modularity of application through separating distribution and adaptability concerns into aspects, which is completely separated of the core system code. Separation makes maintenance and extensibility of the system easy to manage – changing the communication API has no impact in the system code. Adapting the communication protocols to a new distributed middleware requires to rewrite or to add new distribution aspects, which respect API of new communication middleware. Distribution aspects make middleware transparent for the application. Object-oriented techniques don't provide appropriate exception handling mechanism of the distributed application. As a structural part of the communication aspects is a trusted secure connection, which provides privacy of the user communication. Secure aspects makes application independent from used secure protocols.

Exception handling concern allows exception handling to be modified without impacting original system source code as well distributive and adaptive aspects.

Testing is an important process during software development. Separation of concerns AOP provides allows tests to be done and design separately just for functional requirements. Proposed architecture imposes following structure of the software team : "core functionality developers", "test designers" (they don't need to be familiar with AOP), and "aspect developers", which are responsible for the aspect tests and implementation.

In addition proposed architecture allows incremental implementation of application. Part of the functionality could be implemented and tested and then mentioned aspects can be elaborated as well application can built from existing (third party) components which are tied in final application by the adaptive and distributed aspects.

REFERENCES

[1] Aksit, M and M. Bergmans, Composing Synchronization and Real-Time Constraints. Journal of Parallel and Sdistributed Computing 36, pp. 32-52, 1996

[2] Aksit, M., B. Tekinerdogan, and L. Bergams. Achieving adaptability through Separation and Composition of Concerns. In max Mmuhlhauser editor, Special Issues in Object-Oriented Programming, ECOOP'96, Dpunkt-Verlag, 1997.

[3] Birov, D. V. Zhelyazkov, Mobile Banking - Interdisciplinary Approach, Proceedings of 30th Spring Conference of Union of Bulgarian Mathematicians, Borovetz, April 8 - 11, 2001, pp. 253 – 257

[4] Birov D., Aspects, Patterns And Component Models Orthogonality, National Conference with International Participation "Electronica 2004", Sofia, Bulgaria, May 21-22, 2004.

[5] Bollert, K. On Weaving Aspects. Position paper at the ECOOP'99 workshop on Aspect-Oriented Programming, June 1999.

[6] Cugola, G., C. Ghezzi and M. Monga}, Coding Different Design Paradigms for Distributed Applications with Aspect-Oriented Programming}, Workshop su Sistemi Distribuiti: Algoritmi, Architetture e Linguaggi WSDAAL, L'Aquila, Italy, 1999

[7] Cohen, G., J. Chase, and D. Kaminsky. Automatic program transformation with JOIE. In Proceedings of the USENIX 1998 Annual technical Conference, pp. 167-178, Berkeley, USA, June 15-19 1998.

[8] Dantas, A, P. Borba. Developing Adaptive J2ME Application using AspectJ

[9] Dantas, A, P. Borba. Adaptability Aspects: An Architectural Pattern for Structuring Adaptive Applications with Aspects, Third Latin American Conference on Pattern Languages of Programming Sugarloaf-PloP 2003,

[10] Elrad, T., R. E. Filman, and At. Bader. Aspect-Oriented Programming. Communications of the ACM, 44(10):29{32, October 2001.

[11] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1996.

[12] Hannemann, J., and G. Kiczales. Design pattern implementation in Java and AspectJ. In Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications, pp. 161–173. ACM Press, 2002.

[13] Keller R. and Urs Holzle, Binary Component Adaptation. In Eric Jul, editor, ECOOP'98 – Object-oriented Programming, LNCS 1445, pp. 307-329, Springer, 1998.

[14] Kiczales, G. *Aspect-Oriented Programming*. 1997. Procedings of ECOOP, Springer Verang. LNCS 1241.

[15] Kiczales, G. AspectJ: Aspect-oriented Programming using Java Technology. JavaOne Conference, June 2000.

[16] Kiczales, K., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin, Aspect-oriented programming, In Mehmet Aksit and Satoshi Matsuoka, editors, ECOOP'97 – Object-Oriented Programming 11th European Conference, Finland, LNCS 1241, pp. 220-242, Springer-Verlag, New York, NY, June 1997.

[17] Kennens, P., S. Michiels, F. Matthijs, B. Robben, E. Truyen, B. Vanhaute, W. Joosen, and P. Verbaeten. An AOP Case with Static and Dynamic Aspects. In ECOOP'98 Workshop on Aspect-oriented Programming Brussel (Belgium), July 1998.

[18] Lopes, C. D: A Language Framework for Distributed Computing. Ph.D. Disertation, College of Computer Science, Northeastern University, Boston, 1997.

[19] Lyytinen, K., and Y. Yoo. Issues and challenges in ubiquitous computing: Introduction, *Communications of the ACM*, 45(12):62–65, 2002.

[20] Mezini, M., and K. Ostermann, Object Creation Aspects with Flexible Aspect Deployment.

[21] Oreizy, P. , M. M Gorlick., R. N. Taylor, D. Heimbigner, G. Jonhson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approachto self- adaptive software. IEEE Intelligent Systems, 14(3):54–62. (1999).

[22] Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann. Pattern{Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects. Wiley & Sons, 2000.

[23] Soares, S. and P. Borba, OaDA: A Pattern for Distribution Aspects, proceedings of the SugarloafPLoP Conference pp. 87 – 99, 2002

[24] Tarr, P. and H. Ossher. Hype/J user and Installation manual, IBM T. J. Watson Research center, Yorktown heights, NY, USA, 2000

[25] Tarr, P. H. Ossher, W. Harrison, and S. Sutton. N Degrees of separation: Multi-dimensional Separation of Concerns. In ACM, editor, Procedings of the 1999 International Conference onn Software Engineering, pp. 107-119, Los Angeles, CA, USA, 1999.

[26] Waldo, J., G. Wyant, A. Wollrath, and S. Kendall, \A note on distributed computing," in Mobile Object Systems, vol. 1222 of Lecture Notes in Computer Science, pp. 49-64, Springer-Verlag, Berlin, 1997.

[27] Yoder, J.. W. and R. Johnson. The adaptive object-model architectural style. In Working IEEE/IFIP Conference on Software Architecture 2002(WICSA), Montreal, Quebec, Canada, August 25-31 2002.

[28] Zhang, Ch., H. Jacobsen. Aspectizing Middleware Platforms, technical report, Univ. of Topronto, Computer Systems Research |Group, CSRG-466, Jan. 2003

ABOUT THE AUTHOR

Ass.Prof. Dimitar Birov, PhD, Faculty of Mathematics and Informatics, Sofia University, Phone: +359 2 8161 510, E-mail: birov@fmi.uni-sofia.bg.