# Dynamic Service Adaptation for Runtime System Extensions

Robert Hirschfeld, Katsuya Kawamura, and Hendrik Berndt

DoCoMo Communications Laboratories, Future Networking Lab,
Landsberger Strasse 308-312, 80687 Munich, Germany
{hirschfeld, kawamura, berndt}@docomolab-euro.com

**Abstract.** Most of all software systems have to be changed after their initial deployment. This is not only because of changing knowledge and expectations about our domains and systems, but also because of the continuous change of the environment itself. While changes in the environment happen implicitly, we need to explicitly keep our technology in sync with the changing world around it. This is especially true for next generation mobile communication systems which we expect to be open to third-party service providers, allowing them to offer services on a variety of service platforms. Not all of these services to be offered will match with all of the platforms. Adjustments and extensions need to be made to offer a pleasant service experience. Research on dynamic service adaptation provides concepts and technologies needed to perform such changes late in a system's lifecycle, possibly on demand, at runtime, without disruption of service.

## 1 Introduction

Our research at DoCoMo Euro-Labs is directed towards mobile communications technologies beyond the third generation (B3G) that can respond to the requirements of a highly developed multimedia age. B3G systems are expected to not only integrate several access technologies, but also to promote a significant wealth of services offered by a multitude of service providers. In addition to seamless and secure access to heterogeneous networks, B3G systems are considered to encompass high service availability and best service quality to the end user. System requirements are highly demanding. Some of the key requirements essential to B3G communication platforms are to shorten development and provisioning cycles, to minimize system downtimes, to support runtime updates and upgrades, to allow for third-party service integration, and to assist in service personalization.

The unanticipated nature and complexity of forthcoming services and applications makes the support of dynamic service adaptation (DSA) and unanticipated software evolution (USE) inevitable. We regard DSA to be part of the foundation to address phenomena of USE. DSA is motivated by our continuously changing environment, a heterogeneous service landscape, as well as an open system infrastructure. Major goals of DSA are to enable service and platform evolution, to support the advancement of individual parts at a different pace, and to facilitate personalization, context-awareness, and ubiquitous computing. Mobile communication systems that

R. Battiti, M. Conti, and R. Lo Cigno (Eds.): WONS 2004, LNCS 2928, pp. 227–240, 2004.

can be described as long-lived, continuously running, highly-available, embedded, or large-scale widely distributed are most suitable candidates to benefit from DSA.

Most of the adaptation mechanisms deployed today concentrate on content, a few on communication, but almost none on service logic or behavior itself. Thus, content as well as communication adaptation is understood much better than that of service logic or service behavior. In this paper the terms service adaptation, service logic adaptation, or service behavior adaptation are used interchangeably. In contrast to more traditional approaches, we combine aspect-oriented programming with computational reflection and late binding to adapt services and service platforms when changes actually require doing so, as late as possible, if possible without disruption of service.

In this paper we give an overview on our research on software engineering principles and mechanisms for DSA allowing us to evolve, adapt, and extend services dynamically to better support seamless service provisioning and application integration for the next generation mobile communication systems. Our work is aligned with active research in the field of aspect-oriented software development (AOSD, [2]) and USE. We point out how the development of mobile telecommunication systems can benefit from the deployment of AOSD and the provisioning for USE.

The paper is organized as follows: Section 2 illustrates our approach to DSA, addressing modularity and variation points, aspect-oriented programming, late binding and reflection. It also gives an overview of our research platform. Section 3 demonstrates DSA applied in the context of runtime system integration and extension. Section 4 outlines further opportunities for DSA. Section 5 concludes the paper.

## 2 Dynamic Service Adaptation

The concept of adaptability is closely related to that of modularity and variation points. The modularization of a system can improve its flexibility and comprehensibility, and with that can also shorten its development time. Variation points provide us a way to explicitly designate module boundaries in a system's design where changes are expected to happen. The introduction of variation points and with it the separation and composition of common and variable system aspects can provide for flexibility.

The majority of recently built systems are based on object-oriented technologies. Here, classes and instances are employed as both modularity constructs and units of change. Besides other important properties, most aspect-oriented programming (AOP) technologies provide a new, finer grained, modularity construct that allows us to represent crosscutting concerns, down to the methods of individual instances.

Since many changes happen after a system's initial deployment, they need to be addressed very late in its lifecycle. To avoid system downtimes, many of the corrective actions covering these changes need to be performed on demand at runtime. We consider reflective architectures and late binding to be key elements of a DSA platform addressing these requirements. In our approach to DSA, we use the aspect modularity construct to adequately represent units of change. Computational reflection, dynamic AOP, and late binding will allow us to adapt service and service

platforms as late as possible, preferably without system downtimes and with that the disruption of service [11].

In the following subsections we give an overview of modularity, variation points, AOP, reflection, and late binding. The last subsection outlines our research platform for DSA and runtime system extensions.

### 2.1 Modularity, Variation Points, Objects, and Aspects

One approach to manage complexity is modularity. Here, we are trying to improve the comprehensibility and flexibility of a system by decomposing a complex system into smaller, less complex subsystems and then recomposing these subsystems in a principled way. Modules help to hide from each other complex design decisions or design decisions which are more likely to change [22]. Variation points, or hotspots [23], designate module boundaries in a system's design where changes are expected to happen without the need to explicitly name all of them. With variation points we improve flexibility in the context of change through the separation and composition of common and variable aspects of our system.

Variations and variation points depend to a large extent on the underlying modularity mechanism provided by the programming platform a system is built on. Most modern software systems were built using object-oriented technologies where the modularity constructs, and with that the units of change, are that of classes and instances. Here, classes capture the properties of their instances. Although this level of granularity is sufficient in some cases, a more fine-grained approach to modularity is desirable to permit the change of even smaller semantic units such as method implementations.

In object-oriented systems there is code that, even though it implements one particular concern, is spread around (scattered) over many or even almost all modules, crosscutting various other modules implementing other concerns as well, instead of being confined to one or a small number of modules. Because of its non-explicit structure, such crosscutting code is hard to comprehend and difficult to change. The consistency of changes is both hard to verify and to enforce. Object-orientation and its class modularity construct, while proven to be appropriate for many modeling scenarios, cannot be of help in implementing other concerns in a modularized way. Also, while traditional modules such as classes and instances might support the proper structuring of the initial system, subsequent changes to this system could crosscut these module boundaries to affect more than one location.

Based on the assumption that crosscutting is inherent to complex software systems, AOP ([6, 16]) as a new software technology addresses the issues of separation of concerns (SOC, [5, 12]). For that, AOP introduces orthogonal units of modularity to capture crosscutting structures explicitly. Such structures are called aspects and can be found in a software system's requirements, its design, as well as in its implementation. AOP builds on existing technologies but provides additional mechanisms that make it possible to affect a system's implementation in a crosscutting way [4]. Aspects are units of modularity that represent implementations of crosscutting concerns. Aspects associate code fragments (code to be executed when a join point is encountered) with join points (well-defined points in the execution of code) by the use of advice constructs. A collection of related join points descriptors, to be addressed by an advice, is called a pointcut. Join point descriptors denote targets

for the weaving process to apply changes to the underlying computational base system as stated in the advice constructs.

Aspects and their advice are integrated into the base system during an activity called weaving. Weaving in general can be performed at almost any point in time in a software system's lifecycle. Most of today's AOP technologies limit themselves to either compile-time, load-time, or runtime. AspectJ [15] and HyperJ [25] are examples for compile-time weaving. In AspectJ for example, the weaver parses an AspectJ program, transforms the AspectJ abstract syntax tree (AST) into a valid Java [8] AST, and then generates Java byte code for a standard Java virtual machine. JMangler [18] performs load-time transformation of Java class files. AspectS [9] employs run-time weaving to transform the base system according to the aspects involved. The woven code is based on method wrappers [3], reflection [20, 24] and meta-programming [17].

As of today there are several approaches supporting aspect-oriented concepts, ranging from domain-specific aspect languages such as RG [21] or D [19] to general-purpose aspect languages like AspectJ or AspectS. Many of these languages allow us to express crosscutting concerns, down to the level of individual instances, methods and variables. Like objects in object-oriented software development, aspects may appear at all stages of the software development lifecycle. Illustrative examples of aspects that can be commonly observed are architectural or design constraints, features, and systemic properties or behaviors.

### 2.2 Late Binding and Computational Reflection

Software development is still hard. During the software development lifecycle we quite frequently find out something we wished we had known from the very beginning of the project [14]. While there is always a chance that some of the requirements were not sufficiently understood to adequately address them in the software system, many changes happen after a system's initial deployment, and so are impossible to predict and dealt with right from the beginning. On the contrary, such changes must be addressed very late, after deployment. System downtimes can be minimized if most corrective measures can be applied at runtime. To address this requirement, we consider late binding and reflective architectures to be key elements of a DSA platform.

Late binding is a mechanism to defer decisions to a later point in time which allows us to avoid too early commitments to design decisions, especially decisions regarding variation points, we might or will not be able to maintain. Whereas early binding requires us to provide abstractions addressing possible change at a very early point in time, late binding helps us to avoid such premature abstractions. Extreme late binding allows these decisions to be made as late as possible, at runtime.

Systems with reflective architectures incorporate structures representing aspects of themselves [20]. The aggregate of theses structures is called the system's self representation which allows the system to both observe its own computation as well as influence or change it. The activity of observing oneself is called introspection, the activity of changing oneself intercession. For service adaptation, introspection will allow us to observe computational properties of a deployed set of services as well as the computational environment they are running in. Intercession can be based on our observations and result in the alteration of the service.

### 2.3 Adaptation Platform

While most of today's adaptation mechanisms focus on content and a few on communication, almost none considers the adaptation of service logic itself. Because of that, our research on DSA is directed towards behavior adaptation at runtime. To adequately address change, services and service platforms need to be adaptable, as late as possible, when changes actually require adaptation to happen. Making changes effective dynamically at runtime will offer the benefit of avoiding system downtimes, and with that the disruption of service.

Our DSA research platform is based on a layered system architecture. Squeak/Smalltalk serves us a very dynamic object-oriented multimedia scripting environment [7, 13]. AspectS extends the Squeak/Smalltalk environment to allow for experimental aspect-oriented system development. PerspectiveS builds on AspectS to allow for dynamic behavior layering in the Squeak environment.

Squeak/Smalltalk's properties that are important to our research on DSA are its extensive reflection support covering both introspection and intercession, its powerful metaobject protocol [17] that gives us full access to the computational properties of our environment, and its support for very late binding to defer binding decisions until the point when they actually need to be made. The idea of metaobject protocols is that one can and should open languages up to allow users to adjust the design and implementation to make the language or environment to suit their particular needs.

AspectS provides a platform for the exploration of aspect-oriented software composition in the context of dynamic systems [9]. It allows for convenient meta-level programming, addressing the tangled code phenomenon by providing aspect modules. AspectS shows great flexibility by not relying on source or bytecode code transformations. Instead, it makes use of metaobject composition. In contrast to most other approaches to AOP that only focus on class-level aspects, AspectS allows for instance-level aspect and with that for modularization of behavior that crosscuts individual instances.

PerspectiveS coordinates the activation of a set of aspects, and so lets us to decorate a system with context-dependent behavior, without requiring developers of the base system to be aware of that [10]. PerspectiveS enables greater separation of concerns of a base system from its context-dependent behavior. Here, base systems can be freed from providing behavior that explicitly takes action in response to context changes not known at neither development- nor deployment-time. PerspectiveS facilitates basic role modeling by dynamic composition of multiple roles without the loss of object identity. Roles can be added or removed on-demand, with each role bringing in its own set of state and behavior.

All of these layers allow us to both implement our basic service logic as well as to adapt this service logic to additional requirements and unforeseen circumstances if necessary. Due to the dynamic nature of our research platform, adaptation activities can be carried out on an on-demand basis, during runtime, while our services are already deployed and activated [11].

In the following section, we use a scenario of runtime system extension to illustrate the application of our DSA platform.

## 3 Runtime System Extensions

Next generation mobile communication systems will give third-party service providers more opportunities to offer their service on a variety of open service platforms. Since there will be several such platforms, services are likely to not match all of them in the same way: While some services and some platforms are perfect matches, an many cases there is some work to be done to integrate them adequately to ensure an pleasant service experience. As already stated, it is not possible to identify and apply all of theses changes upfront, right from the beginning. Most of them are to become effective after the initial deployment a service or its service platform. And preferably all of them should be applied without noticeable disruption of service.

Our work on DSA not only covers the design and implementation of an adaptation platform, but also includes the illustration of our approach by describing candidate scenarios of DSA. In [11] we explain the application of our DSA platform to integrate a third-party component, the Fauré personal digital assistant (PDA) [1], and the value of DSA by discussing four scenarios: The introduction of additional safeguards let us correct the wrong assumption of the Fauré component provider that this PDA component would be operated standalone and terminating it requires quitting the underlying platform, too. The enforcement of style guide elements allowed us to change the original appearance of the user interface (UI) to conform to the requirements of a particular style guide – be it because of a difference in the style offered by the component and the style required by the platform operator, or because the style guide of the platform operator was changed itself and all existing components conforming to the previous style guide now have to conform to the new one. Late UI branding let us decorate suitable UI elements with brand names, logos, or advertisements. In the category of upgrades, updates, and fixes we resolved an issue with the rendering engine of our platform we discovered while carrying out our late UI branding adaptation.

In the following we will show how to take advantage of DSA to extend the Fauré PDA component with another service application, and to instrument the newly integrated application with a notification mechanism to indicate proper usage indication events.

### 3.1 Our Base Application

We will use the Fauré PDA [1] as the service application to be extended dynamically at runtime. Fauré, an open source PDA implementation for Squeak designed to run on a handheld device, runs on top of our DSA platform, most likely on a mobile terminal.


**Fig. 1.** Fauré Welcome Screen

When we launch our PDA application, its welcome screen shows summarized list of our things to do and our personal schedule (Fig. 1). Via the view menu, we can reach to our full contacts database, all of our social events and things to do, a little sketch pad, a piano-like music instrument, and a demonstration of the 3D rendering facilities of Squeak.

### 3.2 Tetris

The Fauré PDA also provides a game called Same Game, originally written by Eiji Fukumoto for UNIX and X. The object of SameGame is to maximize the score by removing tiles from the board. Tiles are selected and removed by clicking on a tile that has at least one adjoining tile of the same.

But what if most of our customers would like to play another more popular game, a game like Tetris? Tetris was originally developed by Alexey Pazhitnov on an Electronica 60. In Tetris, regularly-shaped blocks appear at the top of the screen and advance steadily down a fine grid. These blocks can be spun to make them fit into point-scoring rows. As levels get completed, Tetris is getting faster what makes it harder to spin and fit blocks together to complete the rows.

### 3.3 Tetris Integration

After searching for an implementation of Tetris, we find one the runs in our execution environment (Fig. 2). Unfortunately, that implementation does not fit into our PDA: The UI element representing the game is too large because its height exceeds the height made available for user applications by the PDA. Also, the game control buttons that allow us to rotate and drop Tetris pieces are in a location that would cause us to waste even more screen real estate we cannot afford.

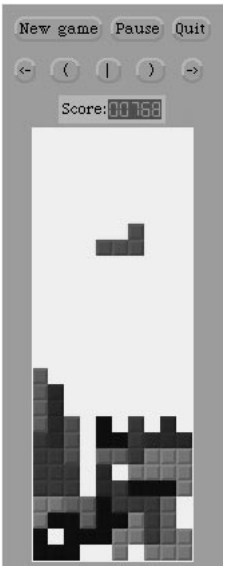


**Fig. 2.** Tetris

A common approach to make the new Tetris game fit into the PDA environment would be to obtain its source code, change this source code, and completely rebuild the game application. Another way to make Tetris conform to our requirements is to provide an additional piece of software that instructs our runtime environment on how to transform this game to become deployable within our provisioning environment.

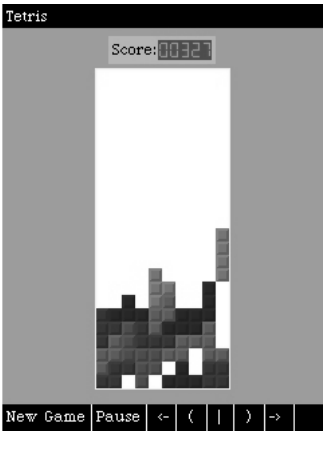


**Fig. 3.** Adapted and Integrated Tetris

Fig. 3 shows the same Tetris applications previously discussed after its transformation and integration into the PDA. One can see how its size was changed to meet the constraints imposed by the PDA. Also, all game control buttons previously

found on top of the game area are now arranged in the bottom row of the PDA UI where one would have placed them in the first place if the game would have been designed to run in the PDA from the beginning.

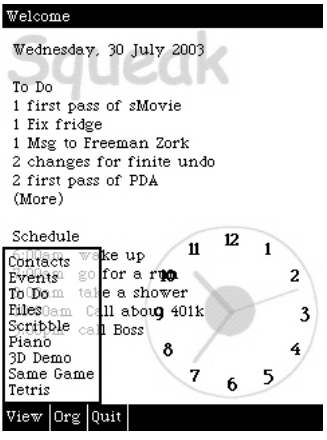


**Fig. 4.** Fauré View Menu with Tetris Menu Entry

Making Tetris fit is not enough to claim its integration is done. It needs to be accessible by the user, too. For that we have to extend the launch menu of our PDA by providing an entry that will launch Tetris if selected. Fig. 4 shows the extended menu, with our new Tetris entry last in the list.

### 3.4 Usage Indication and Metering

Merely providing new applications and services to our customers might not be sufficient from a business' point of view. Providing services implies most of the time some form of compensation, either directly or indirectly. Compensation is typically based on service level agreements (SLAs) about quantitative information about the usage of a service. Since most of the time third-party software components are not developed to target specific SLAs and also because SLAs can change as often as possible, it is not of benefit to commit to specific usage indications too early in the service lifecycle.

DSA allows us to instrument our applications and services to provide usage indication information, not even after their development, but also after their deployment, as late as at runtime.

Fig. 5 shows a usage indication trace of Tetris, where each start of a new game is reported to usage collection mechanisms which can act as an input feed to a rating and billing engine. This usage indication record generation was introduced by one of our adaptation modules that instrument the original Tetris component.

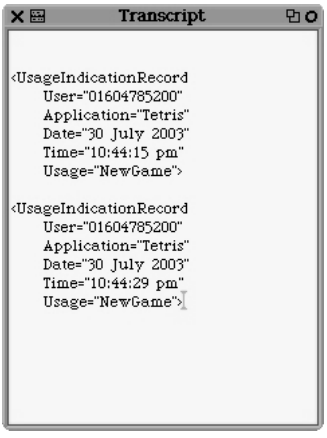


**Fig. 5.** Posted Tetris Usage Indication Records

The following listings illustrate how this adaptation was achieved. The first listing shows the method that gets invoked every time a customer presses the 'New Game' button (`TetrisBoard>>newGame`): Tetris starts over with a new game.

```
TetrisBoard>>newGame
    self removeAllMorphs.
    gameOver _ paused _ false.
    delay _ 500.
    currentBlock _ nil.
    self score: 0.
```

In the next listing we can see code that belongs to our adaptation module (`FdsaTetrisUsageAspect`) and is responsible for instrumenting the `newGame` Method in such a way that every time (except for the first) it gets invoked, a usage indication record will be posted to the responsible entity (in this simplified case the system transcript, Smalltalk's console).

```
FdsaTetrisUsageAspect>>adviceTetrisBoardNewGame
 ^ AsBeforeAfterAdvice
    qualifier: (AsAdviceQualifier
       attributes: { #receiverClassSpecific. })
    pointcut: [OrderedCollection
       with: (AsJoinPointDescriptor
          targetClass: TetrisBoard
          targetSelector: #newGame)]
    afterBlock: [:rcvr :args :aspect :client :return |
       thisContext baseSender baseSender selector
          ~~ #initialize "the first game is for free"
             ifTrue: [self postTetrisUsage]]
```

The convenience method `postTetrisUsage` is implemented as follows:

```
FdsaTetrisUsageAspect>>postTetrisUsage
    Transcript
       cr; show: '<UsageIndicationRecord User="',
```

```
self userIdentifier printString,
'" Application="Tetris" Date="',
Date today printString, '" Time="',
Time now printString, '" Usage="NewGame">'.
```

Our deployed PDA service will be accompanied by the Tetris component to be integrated and the adaptation modules necessary to do so. The adaptation module shown above is only responsible for dynamic usage indication record generation, the adaptation module required to integrate Tetris into the PDA service is not shown in this paper.

## 4 Further Opportunities

There are quite a few opportunities for DSA and runtime system extensions. The following subsections will illustrate how personalization, application-level security, pre-standard releases, and addressing regulatory requirements can benefit from DSA.

### 4.1 Personalization

Personalization is regarded to be one of the most compelling features for mobile communications systems B3G by supporting users in selecting the best services from the rapidly increasing diversity of mobile services, and adjusting selected services to their individual needs. Service personalization promises to foster and improve the relationship between service providers, mobile operators and customers. Also, it is expected to promote the adoption of increasingly complex services.

Service personalization can basically be approached from two points of view. On the one hand, there is the user perspective where user models are developed and expressed through user profiles and user preferences within the respective system of client devices, services, and applications. We consider context awareness to be an integral part of this user-centered position.

On the other hand, there is the system side where we need to consider how personalization features are implemented and how the personalization of mobile applications effects actual system execution and runtime behavior. An example is the impact of changes in a user profile on the service delivery in a given situation, for instance taking the change of a user's geographical location into consideration.

Service personalization is not limited to data (such as selective content delivery) or the user interface only, but will also involve DSA with changes to behavior (service logic) and interaction (service signaling and communication). Such service adaptation allows mobile systems to react to changes in the environment, which are inherent in their nature to users roaming in a federated world-wide service space. For instance, a personalized software application may be downloaded by users, based on their personal preferences or current environments. To implement this, the appropriate user aspects have to be merged for a personalized service.

DSA also supports the creation of services being capable of dynamic personalization. For example, adaptation of service behavior can be made possible

through extensions to the service provisioning infrastructure that allows the selection of units of modularity to be adjusted.

### 4.2 Application-Level Security

One of the main acceptance criteria for new communication and collaboration services is an adequate management of privacy. We need to ensure all privacy policies and security constraints to be enforced consistently across the whole system. Furthermore, in an open environment, we need to assure that our security restrictions and privacy policies not only affect components currently installed and running, but also the ones that will be installed in the future.

To ensure that, we need a mechanism that continuously observes the runtime platform and adjusts to the requirements all newly added components in a consistent manner. DSA can play an important role in providing such a mechanism.

### 4.3 Pre-standard Releases

Very often, standardization processes take a long time, and most of the time longer than expected. While shipping standard conformant products is essential for solutions that have to be integrated with a heterogeneous environment, time to market is most of the time more critical to the success of a business than standard conformance.

DSA allows for both, early product releases and standard conformance. If early releases of a standard become reasonably stable, affected component can be released at that time. Once the final release of the standard becomes available, all affected and already deployed components can be updated to conform to the available standard. Advanced product planning will be possible through the application of DSA in later phases of the lifecycle of a product.

### 4.4 Regulatory Requirements

The same said about pre-standard releases holds for regulatory requirements to be met. Whenever changes of laws or other regulations affect products and systems already released and deployed, such products and systems need to be adjusted. This process can become very cost intensive if carried out the traditional way by building a completely new system, taking down the old systems and bringing up the new ones, possibly with the consequence of service outages and all economical consequences involved.

DSA allows us to upgrade deployed and running systems, at runtime, without the need to disrupt any service provided. Delta modules can provide the additional or changed functionality needed to meet new requirements, and the DSA infrastructure makes these modules effective without service disruption if possible.

## 5 Summary

We expect next generation mobile communication systems to be more open to third-party service providers, yielding a rich and flexible service landscape. With that, such systems will be more complex than ever before. Different parts of the system will evolve at a different pace. Service offerings continuously come and go. And because change is rather the norm than the exception, service platforms need to prepare for it. Instead of relying on premature abstractions, other mechanisms are required to allow for system adaptations to be performed – when they are needed, on-demand. To ensure a pleasant service experience and to avoid system downtimes and disruptions of service as much as possible, necessary adaptations should preferably carried out during runtime. In this paper we show what we believe is necessary to dynamically adapt services by giving an overview of our approach, our adaptation platform, and by showing how to apply these concepts and technologies to integrate and extend services at runtime. While in the past most of the adaptation strategies are based on redundancy and failovers, this is no longer possible anymore in a world of small mobile devices. A new approach is required to deal with change. DSA is ours.

**Acknowledgements.** We would like to thank Matthias Wagner, Stefan Hanenberg, Andreas Raab, Wolfgang Kellerer, Anthony Tarlano, and Christian Prehofer for their contributions.

## References

1. Allen, R.: *Faure*. http://russell-allen.com/squeak/faure/.
2. Aspect-Oriented Software Development homepage (http://www.aosd.net/).
3. Brant, J.; Foote, B.; Johnson, R.; Roberts, D.: *Wrappers to the Rescue*. In: Proceedings of the 1998 European Conference on Object-Oriented Programming (ECOOP), pp. 396–417, Brussels, Belgium, 1998.
4. Elrad, T. ;Aksit, M.; Kiczales, G.; Lieberherr, K.; Osher, H.: Discussing Aspects of AOP. In: *Communications of the ACM*, Vol. 44, No. 10, pp. 33–38, October 2001.
5. Ernst, E.: *Separation of Concerns*. In: Proceedings of the AOSD 2003 Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT), Boston, MA, USA, March 2003.
6. Filman, R.E., Friedman, D.P.: *Aspect-Oriented Programming is Quantification and Obliviousness*. In: Proceedings of the ECOOP 2001 Workshop on Advanced Separation of Concerns, Budapest, Hungary, June 2001.
7. Goldberg, A.; Robson, D.: *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
8. Gosling, J.; Joy, B.; Steele, G.; Bracha, G.: *The Java Language Specification (Second Edition)*. Addison-Wesley, 2000.
9. Hirschfeld, R.: *AspectS – Aspect-Oriented Programming with Squeak*. In: M. Aksit, M. Mezini, R. Unland, editors, Objects, Components, Architectures, Services, and Applications for a Networked World, LNCS 2591, pp. 216–232, Springer, 2003.
10. Hirschfeld, R.; Wagner, M.: *PerspectiveS – AspectS with Context*. In: Proceedings of the OOPSLA 2002 Workshop on Engineering Context-Aware Object-Oriented Systems and Environments (ECOOSE), Seattle, WA, USA, 2002.

11. Hirschfeld, R.: *Dynamic Service Adaptation*. DoCoMo Euro-Labs Technical Report, ITR-FNL-023, Munich, April 2003.
12. Hürsch, W.L.; Lopes, C.V.: *Separation of Concerns*. College of Computer Science, Northeastern University, Boston, USA, February 1995.
13. Ingalls, D.; Kaehler, T.; Maloney, J.; Wallace, S.; Kay, A.: *Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself*. In: Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pp. 318–326, Atlanta, GA, USA, October 1997.
14. Kay, A.: *Is "Software Engineering" an Oxymoron?* Viewpoints Research Institute, 2002.
15. Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G.: *An Overview of AspectJ*. In: Proceedings of the 2001 European Conference on Object-Oriented Programming (ECOOP), pp. 327–355, Budapest, Hungary, 2001.
16. Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, Ch.; Lopes, C. V.; Loingtier, J.-M.; Irwin, J.: *Aspect-Oriented Programming*. In: Proceedings of the 1997 European Conference on Object-Oriented Programming (ECOOP), pp. 220–242, Jyväskylä, Finland, 1997.
17. Kiczales, G.; des Rivieres, J.; Bobrow, D.: *The Art of the Metaobject Protocol*. Addison-Wesley, 1991.
18. Kniesel, G.; Costanza, P.; Austermann, M.: *JMangler – A Framework for Load-Time Transformation of Java Class Files*. In: Proceedings of the Workshop on Source Code Analysis and Manipulation (SCAM). Florence, Italy, November 2001.
19. Lopes, C. V.: *D: A Language Framework for Distributed Programming*. Dissertation. College of Computer Science, Northeastern University, Boston, USA, 1997.
20. Maes, P.: *Concepts and Experiments in Computational Reflection*. In: Proceedings of the 1987 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 147-155, Orlando, FL, USA, 1987
21. Mendhekar, A.; Kiczales, G.; Lamping, J.: *RG: A Case-Study for Aspect-Oriented Programming*. Xerox PARC. Technical Report SPL97-009 P9710044. February 1997.
22. Parnas, D.L.: On the Criteria To Be Used in Decomposing Systems into Modules. In: *Communications of the ACM*, Vol. 15, No. 12, pp. 1053–1058, December 1972.
23. Pree W.: *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.
24. Rivard, F.: *Smalltalk: A Reflective Language*. In: Proceedings of Reflection 1996.
25. Tarr, P.; Ossher, H.; Harrison, W.; Sutton Jr., S.M.: *N Degrees of Separation: Multi-Dimensional Separation of Concerns.* In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 107–119, Los Angeles, CA, USA, May 1999.