Dynamic Service Adaptation

Robert Hirschfeld and Katsuya Kawamura
DoCoMo Euro-Labs
Future Networking Lab
Landsberger Strasse 308-312
80687 Munich, Germany
(hirschfeld|kawamura)@docomolab-euro.com

Abstract – The only constant is change. Change can be observed in our environment, in the technology we build, and in ourselves. While changes in the environment happen continuously and implicitly, our technology has to be kept in sync with the changing world around it. Although for some of the changes we can prepare, for most of them we cannot. This is especially true for next generation mobile communication systems that are expected to support the creation of a ubiquitous society where virtually everything is connected and made available within an organic information network. Countless resources will frequently join or leave the network, new types of media or new combinations of existing ones will be used to interact and cooperate, and services will be tailored to preferences and needs of individual customers to better meet their needs. Research in the area of dynamic service adaptation will provide concepts and technologies required to make such a dynamic environment a reality.

I. INTRODUCTION

Mobile communication systems beyond the third generation (B3G) are not only expected to integrate several networks, but also to encourage a substantial richness of services through third-party service offerings. In this context, comprehensive support of complex and dynamic computing environments, distributed over multiple service platforms, is essential to adequately address high demands of mobile multimedia services B3G. The unanticipated nature and complexity of forthcoming services and applications makes support of dynamic service adaptation (DSA) and unanticipated software evolution (USE) a necessity.

In this paper we give an overview of some of our research efforts towards systems B3G. These include software engineering principles and mechanisms for software evolution in mobile communications systems and dynamic adaptation for service integration and personalization. We align our work with active research in the field of aspect-oriented software development (AOSD, [2]) and USE and point out how the development of highly-distributed mobile telecommunication systems can benefit from the deployment of AOSD and USE.

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. With respect to that, system requirements are highly demanding. Here are some of the key issues we have identified to be essential to B3G communication platforms:

- Short development and provisioning cycles,
- Minimal system downtimes,
- Runtime updates/upgrades support,
- Third-party component and service integration,
- Integration of heterogeneous environments, and
- Service personalization.

We consider DSA to be significant in our ability to address these issues, at both the network and the terminal side. The intention of DSA is 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. We consider long-lived, continuously running, highly-available systems which might be embedded or large-scale widely distributed – in other words: mobile communication systems – to be chief candidates to benefit from DSA.

Most of the adaptation mechanisms deployed today concentrate typically on content, not so often on communication, but almost never on service logic or behavior itself. Thus, content as well as communication adaptation is understood much better than service logic adaptation. For convenience, we will use the term service adaptation to denote service logic or behavior adaptation.

In contrast to more traditional approaches, we combine aspect-oriented programming (AOP, [6, 15]) with computational reflection and late binding to adapt services and service platforms when changes actually require doing so, as late as possible, preferably without disruption of service.

The remainder of our paper is organized as follows: Section II illustrates our approach to dynamic service adaptation, addressing modularity and variation points, AOP, late binding and reflection. An overview of our

research platform is given in section III. Section IV demonstrates dynamic service adaptation applied in the context of third-party service integration. Section V concludes our paper.

II. APPROACH

We are concerned with the 'what', 'when', and 'how' of service adaptation. The 'what' of service adaptation distinguishes between the basic properties of software systems – computation, state, and communication. The 'when' of service adaptation addresses the time when adaptations can be made operational in a system – during software development, at compile-time, load-time, or run-time. The 'how' of service adaptation studies tools and techniques that allow for adaptations to become effective.

The concept of adaptability is closely related to that of modularity and variation points. Modularization is a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. Variation points allow us to explicitly designate module boundaries in a system's design where changes are expected to happen without the need of explicitly naming these changes. Variation points are introduced to support flexibility through the separation and composition of common and variable system aspects. Variations and variation points depend on the modularity mechanism provided by the programming platform a system is built on. Most newly built systems are based on object-oriented technologies with classes and instances as modularity constructs as well as units of change. AOP provides a new, more fine grained, modularity construct that allows us to represent crosscutting concerns, down to the methods of individual instances.

Many changes happen after a system's initial deployment and need to be addressed very late in a system's lifecycle. It is preferable, or even required, to avoid system downtimes by performing as many corrective actions on demand at-runtime. To address this requirement, we consider reflective architectures and late binding to be key elements of a platform for DSA.

In our approach to DSA, we use the aspect modularity construct to most adequately represent units of change. Computational reflection, dynamic AOP, and late binding will allow us to adapt service and service platforms when changes actually require doing so, as late as possible, preferably without system downtimes and with that the disruption of service.

In the following subsections we give a brief introduction into the concepts of modularity, variation points, AOP, reflection, and late binding.

A. Modularity and Variation Points

Modularity is one approach to manage complexity. By organizing a complex system into smaller less complex subsystems and then recombining these subsystems in a principled way, we are trying to improve the comprehensibility of a system as well as its flexibility and so reducing its development time [21]. We design modules to hide from each other complex design decisions or design decisions which are more likely to change [21]. Variation points, also called hotspots [22], allow us to designate module boundaries in a system's design where we expect changes to happen without explicitly naming them. With variation points, we gain flexibility in the context of change through the separation and composition of common and variable system aspects.

In object-oriented programming, for example, the basic unit of modularity is that of objects. Class objects capture the properties of their instances. Instead of being localized within one or a small number of modules, code that implements a particular concern is spread around (scattered) over many or even almost all places, crosscutting various other modules implementing other concerns as well. Because of its non-explicit structure, such crosscutting code is difficult to reason about, and with that it is also difficult to change. The consistency of changes is both hard to prove and hard to enforce. Object-oriented programming and its class modularity construct, while proven to be appropriate for many modeling scenarios, cannot be of help in implementing the logging concern in a well modularized way.

Variations and variation points depend 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. 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. 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.

B. Aspect-Oriented Programming

AOP ([6, 15]) is a new software technology addressing the issues of separation of concerns (SOC, [5, 11]). It is based on the assumption that crosscutting is inherent to complex systems. AOP addresses these issues by introducing new/alternate units of modularity to capture crosscutting structures explicitly. Such structures are

called aspects and can be found in a software system's design as well as its implementation.

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. A collection of related join points, to be addressed by an advice, is called a pointcut. Join point descriptors denote targets for the weaving process to apply computational changes to the underlying base system stated in advice objects.

The activity of integrating aspects and their advice into the base system is called weaving. Weaving in general can be performed at compile-time, load-time, or runtime. AspectJ [14] is an example for compile-time weaving. Here, the weaver parses an AspectJ program, transform 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 [17] performs load-time transformation of Java class files. AspectS [9] employs a run-time weaver to transform the base system according to the aspects involved. The woven code is based on method wrappers [3], reflection [19, 23] and meta-programming [16].

As of today there are several approaches that support aspect-oriented concepts, ranging from general-purpose aspect languages like AspectJ or AspectS to domain-specific aspect languages such as RG [20] or D [18]. Many of these languages allow us to represent crosscutting concerns, down to the methods and instance variables level of granularity. Like objects in object-oriented programming, aspects may appear at all stages of the software development lifecycle. Examples of aspects that can be commonly observed are architectural or design constraints, features, and systemic properties or behaviors (such as error recovery and logging).

C. Late Binding and Reflection

During the software development and product lifecycle it happens quite frequently that we find out something we wished we had known from the very beginning of the project [13]. While there is always the 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 with that are impossible to anticipate and address right from the beginning. On the contrary, such changes must be addressed very late, after deployment, during production. System downtimes can be minimized if most corrective actions can be carried out at runtime.

To address this requirement, we consider reflective architectures and late binding to be key elements of a platform for DSA.

Reflective architectures are implemented by systems that incorporate structures representing (aspects of) themselves [19]. The aggregate of theses structures is called the system's self representation which allows the system to both observe its own execution as well as influence or change its own behavior. The former property of a reflective system is called introspection and the latter intercession. In the context of service updates and 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 then can be based on our observations and result in the alteration of the service/system. While there is also research on the subject of compile-time reflection (especially in the context of generative programming, [4]), we are talking about runtime reflection if not explicitly stated otherwise.

Late binding describes a mechanism to defer decisions to a later point in time. With late binding, we can 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.

III. Platform

With DSA we want services and service platforms to be adaptable, as late as possible, when changes actually require adaptation to happen, with the benefit of avoiding system downtimes and with that the disruption of service.

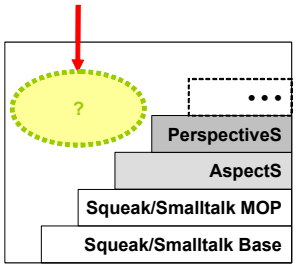

Figure 1. Dynamic Adaptation Platform

To carry out our research, we need to comprehend the nature of fully dynamic systems to advance our understanding of the possibilities as well as the difficulties of our ambitions. The selection and extension of our research platform is an important factor of our making progress. Our platform constituents builds on top of each other, which leads to a layered architecture as depicted in Fig. 1.

Squeak/Smalltalk serves us a very dynamic object-oriented multimedia scripting environment [7, 12]. Some

of its most remarkable properties are its extensive reflection support covering both introspection and intercession, its powerful metaobject protocol [16] that gives us full access to the computational properties of our platform, 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. With that, users are encouraged to actively participate in the language design process. Language designs based on a metaobject protocol are themselves implemented as object-oriented systems which take advantage of object-orientation to make the properties of such language implementations flexible.

AspectS extends the Squeak/Smalltalk environment to allow for experimental aspect-oriented system development. The goal of AspectS is to provide a platform for the exploration of aspect-oriented software composition in the context of dynamic systems. It supports simplified guided meta-level programming, addressing the tangled code phenomenon by providing aspect related modules. AspectS shows great flexibility by not relying on code transformations (neither source nor byte code) but making use of metaobject composition instead. In contrast to most other approaches to AOP that only focus on class-level aspects, AspectS allows for instance-level aspect and with that allowing for modularization of behavior that crosscuts a set of individual instances.

PerspectiveS builds on AspectS to allow for dynamic behavior layering in the Squeak environment. It coordinates context-awareness 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 potential decorations. PerspectiveS enables greater separation of concerns of a base system from its context-dependent behavior [10]. With that, base systems can be relieved from providing behavior that explicitly takes action in response to context changes not known at neither development- nor deployment-time. PerspectiveS facilitates 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.

In the next section, we use the scenario of third-party service integration to illustrate the application of our DSA platform.

IV. Third-Party Service Integration

We expect B3G mobile communication systems to be open to third-party service providers, allowing them to offer their services. Not all services to be offered will exactly match with service platforms operated. Adjustments need to be made to eventually offer a pleasant service experience to customers. While some of the adjustments can be identified and applied upfront, many of them will be required after the initial service deployment, perhaps without disrupting a currently active service.

In the following, we illustrate the value of DSA by discussing four such situations that can occur during third-party service integration in more detail. The situations we selected are:

- Additional safeguards,
- Style Guide Conformance,
- Late UI Branding, and
- Upgrades, Updates, Fixes.

This list is by far not complete. System extensions, usage indication, metering, personalization, per-standard releases, and the meeting of regulatory requirements are prime candidates to be added.

All insights concerning DSA in the context of third-party service integration, safeguard introduction, style guide conformance, UI branding, upgrades, updates, fixes and so on are derived from first-hand experience, gained through prototypical implementations. Here, Squeak, AspectS and PerspectiveS serve as part of our research platform. Squeak provides a reflective runtime environment with late binding and reflection facilities. AspectS adds quantification and obliviousness to dynamic systems. PerspectiveS offers context-aware adaptation activation mechanisms.

We decided to offer a new service called a personal digital assistant (PDA) which is intended to be used by our subscribers at their mobile terminal. The particular PDA implementation we utilize on our service platform is Fauré [1], an open-source PDA implementation designed to run on a handheld device.

Obtaining the third-party component implementing this PDA service was simple. We locate the component in our component repository (the Web), and downloaded and installed it onto our service platform, acting as our integration testbed. The welcome screen of an active Fauré PDA summarizes to-do items and scheduled events at a certain time for a certain day.

We can start using our new PDA right away organizing our list of things to do, our personal schedule, our contact information, or our social events to follow up. Our new

PDA also offers us a little notepad where we can sketch notes or little drawings, a little piano to explore some music, to start some 3D demos that emphasize the power of our new toy, or to just play a game.

A. Additional Safeguards

Until now, we faced no problem integrating the PDA component into our service provisioning environment. Ending the PDA service by pressing the Quit button, however, will reveal an assumption made by the original developer of this third-party component that is not acceptable for our platform: Instead of ending only the PDA service, quitting the PDA will also quit the PDA's execution platform and with that our entire service platform.

To make sure that something like that will not happen in a production environment, we need to adjust the behavior of the Quit button functionality. Instead of asking the original Fauré developers to change their components to fit our needs, and also instead of us performing those changes in the component's source code ourselves, we decide to perform an adaptation in a non-invasive manner (meaning not affecting the original source code of the original implementation) by applying DSA. We provide an adaptation module (an aspect) that will accompany the original component and instruct our runtime environment to insert additional behavior into the Quit button functionality so that every time customers want to end their new PDA service they will be asked if they want to exit only the PDA or the whole runtime platform (Fig. 2).

Figure 2. Additional Safeguard

In this example we introduced an additional dialog to better visualize the change applied. In a commercial system we would most likely not offer such an option, but exit the PDA only without giving the choice to exit our platform (here Squeak) in the first place.

The following two listings illustrate how the adaptation was achieved. Listing 1 shows the method that gets invoked every time a customer presses the Quit button: Our PDA saves its current state, and after that invokes the quit primitive (Smalltalk quitPrimitive) of our platform, with the consequence of terminating the entire platform (Squeak).

```
FaureWorld class>>
quit
    PDA current saveDatabase: 'db.pda'.
    Smalltalk quitPrimitive.
```

Listing 1. Quit Primitive Invocation

Listing 2 shows part of our adaptation module (FdsaQuitAspect) that instruments the previously discussed method. Employing AspectS [9], we construct an advice (AsAroundAdvice) that provides code to be executed instead of the original quit method of FaureWorld: After saving the state of our PDA as in the original implementation, we insert a dialog (self confirm: …) asking our customer if only the PDA is to be terminated or actually the entire platform (Squeak).

```
FdsaQuitAspect>>
adviceBrowserBuildMorphicSystemCatList
    ^ AsAroundAdvice
        qualifier: (AsAdviceQualifier
            attributes: { #receiverClassSpecific. })
        pointcut: [OrderedCollection
            with: (AsJoinPointDescriptor
                targetClass: FaureWorld class
                targetSelector: #quit)]
        aroundBlock: [:receiver :args :aspect :client :clientMethod |
            | ctx morph |
            PDA current saveDatabase: 'db.pda'.
            (self confirm: 'Quit Squeak, too?')
                ifTrue: [Smalltalk quitPrimitive]
                ifFalse: [self deleteFaureWorld]]
```

Listing 2. Safeguard Dialog for Quit

Our deployed PDA service is accompanied by this adaptation module that instructs our service platform to carry out the desired adaptation step.

B. Style Guide Conformance

Many operators require third-party services provided through their infrastructure to conform to specific user interface (UI) style guides. Prominent examples are style guides for i-mode by NTT DoCoMo and for Vodafone live by Vodafone. Non-conformance to such style guides can cause misunderstanding on the user's side and ultimately mean that the service portfolio offered by a service provider is not well-selected. This can and will cause harm to customer acceptance, at best of an individual service offering or of the entire service

portfolio at worst. Style guide related adaptations might not only be necessary for third party components not developed originally with a specific style guide in mind, but also when existing style guides or policies get changed.

In our example, we have chosen a style guide that requires the text that appears on Quit buttons to be rendered using the color red. The developers of the Fauré PDA did not anticipate the color of the Quit button text to be a concern for us to be changed, and because to that they did not provide a means to change it. Instead, the coloring of the Quit button is hidden somewhere in the UI initialization sequence of the PDA component.

Figure 3. Imposed Style Guide

Fig. 3 shows Fauré's UI after it has been adapted by us. We applied a non-invasive adaptation module that changed the color used to initialize the Quit button's text to be red.

```
FaureMenuBar>>
addButton: aName withAction: aSymbol target: aTarget
    | m |
    (m _ SimpleButtonMorph new) label: aName;
        borderWidth: 0;
        target: aTarget;
        actionSelector: aSymbol;
        actWhen: #buttonDown;
        cornerStyle: #square;
        color: Color black;
        height: 20;
        vResizing: #rigid;
        hResizing: #rigid;
        layoutInset: 3;
        changeTableLayout.
    (m findA: StringMorph)
        color: Color white.
    self addMorph: m.
```

Listing 3. Button Initialization

In Listing 4 we can see part of our adaptation module (FdsaQuitButtonMigrateAspect) that instruments Fauré's menu bar button construction method as follows: We create an AspectS advice (AsBeforeAfterAdvice) that provides code to be executed after each invocation of the addButton:withAction:target: method of FaureMenuBar. Our code checks if the button constructed actually is a Quit button, and if so it changes its text color to red (m color: Color red).

```
FdsaQuitButtonMigrateAspect>>
adviceFaureMenuBarAddButtonWithActionTarget
    ^ AsBeforeAfterAdvice
        qualifier: (AsAdviceQualifier
            attributes: { #receiverClassSpecific. })
        pointcut: [OrderedCollection
            with: (AsJoinPointDescriptor
                targetClass: FaureMenuBar
                targetSelector: #addButton:withAction:target:)]
        afterBlock: [:receiver :args :aspect :client :return |
            | m |
            m _ receiver submorphs first findA: StringMorph.
            (m notNil and: [m contents = 'Quit'])
                ifTrue: [m color: Color red]]
```

Listing 4. Specialized Quit Button Initialization

While such a change only becomes effective during the start-up of a PDA component, this style guide related adjustment needs to be applied to all running PDA components also, meaning to PDA components that were already started and with that already run their UI initialization sequence. To achieve that, we greatly benefit from the reflective nature of our runtime platform: We employ a metaprogram that finds all active PDAs not yet conforming to our style guide requirements and transforms all places necessary to make all existing Quit buttons to render their text in red.

C. Late UI Branding

Many third-party components offer UI elements that could be used for additional branding, which could be used by the operator of a service platform or the service providers themselves to place brand names, trade marks, or even advertisements. Unfortunately, most of the time, component providers do not provide explicit interfaces that would allow us to make use of those additional opportunities for branding.

DSA allows us to augment basic UI rendering to place additional branding related information onto UI widgets and other surface areas without anticipated interfaces to do so explicitly.

The Fauré PDA comes with a 3D demo to show the high-performance 3D rendering capabilities of the Squeak environment Fauré makes use of. The demo displays a cube with its six square sides rendered in a different color. Wheel controls allow this cube to be zoomed and rotated in all three dimensions.

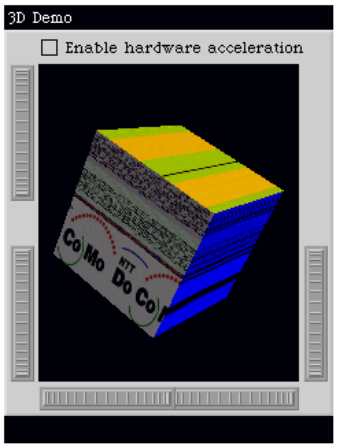

Figure 4. Late UI Branding

Since the surface area of the cube is rendered using a plain texture, it is a prime candidate for additional branding. We provided an adaptation module that places an additional texture, the DoCoMo Euro-Labs logo, at its surface (Fig. 4). The application of our adaptation can again be characterized as dynamic and non-invasive because it can be applied and revoked at runtime, and it does not change the source code of the original component adapted to our current needs.

Listing 5 shows the code used to initialize Fauré's 3D demo scene. Here, a 3D scene object (a cube) is created and added to the actual scene, without providing any specific texture to be rendered on the sides of the cube.

```
B3DSceneMorph>>
createDefaultScene
    | sceneObj camera |
    sceneObj _ B3DSceneObject named: 'Sample Cube'.
    sceneObj geometry: (B3DBox
        from: -0.7@-0.7@-0.7 to: 0.7@0.7@0.7).
    camera _ B3DCamera new.
    camera position: 0@0@-1.5.
    self extent: 100@100.
    scene _ B3DScene new.
    scene defaultCamera: camera.
    scene objects add: sceneObj.
```

Listing 5: Sample 3D Scene Creation

In listing 6 we have another adaptation module (FdsaDcml3dMigrateAspect) creating an advice (AsBeforeAfterAdvice) that adds some code to be executed after the creation of the 3D demo scene in createDefaultScene. Here we provide the 3D demo object with our DoCoMo Euro-Labs logo as new texture.

```
FdsaDcml3dMigrateAspect>>
adviceB3DSceneMorphCreateDefaultScene
    ^ AsBeforeAfterAdvice
        qualifier: (AsAdviceQualifier
            attributes: { #receiverClassSpecific. })
        pointcut: [OrderedCollection
            with: (AsJoinPointDescriptor
                targetClass: B3DSceneMorph
                targetSelector: #createDefaultScene)]
        afterBlock: [:receiver :args :aspect :client :return |
            receiver scene objects first
                texture: ((Form fromFileNamed: 'dcml.jpg')
                    asTexture wrap: true)]
```

Listing 6. Offering a Texture for the 3D Cube

This particular adaptation is yet another example for the need of instance or state migration necessary to adjust existing objects with state that is the result of side effects that have occurred previous to the activation of our adaptation module.

D. Upgrades, Updates, Fixes

By looking at the logo of DoCoMo Euro-Labs in Fig. 4, we discover a 3D rendering bug. This rendering bug is not a bug introduced by Fauré but was already there in our runtime environment. Now that we discovered it, it would also be nice to fix it momentarily, without the need to rebuild the whole system, shutting down all nodes that need to be fixed, replacing the old malfunctioning system with the newly built one, and bringing everything up again. Note that bringing a system down and up again might require us to backup and restore operational state if necessary.

Figure 5. Fixed 3D Rendering Problem

Instead of exercising the procedure of rebuilding and exchanging the system, we provide a dynamic adaptation module that fixes the 3D rendering problem while our system is running (Fig. 5).

V. Summary and Outlook

We believe that next generation mobile communication systems will be more complex than ever before. This is not only caused by the increased complexity of the environment these systems are connected with, but in part also due to the assumption that such systems will be open for third-party service providers to offer their services to end customers using an operator's communications platform. Service providers come and go, their service portfolio is adjusted to the needs and the preferences of their customers, and service level agreements will be one of the key differentiation factors. Because changes are rather the norm than an exception and usually cannot be planned for, we need new concepts, mechanisms and technologies to better support change to make all that possible. We need to investigate what is really required to make us comfortable with change. We then have to understand how to either evolve our computational platforms to meet our needs, to migrate to a different and better platform, or if such a platform does not exist yet, how to build one ourselves. Besides concepts, mechanisms and technologies, we also need appropriate infrastructure support to propagate adaptation modules, to coordinate their activation and deactivation, to detect and resolve conflicts if necessary, and to address safety and security concerns related to mobile code. Change activation and deactivation, or adaptation composition in general, can go beyond a basic approach towards semantic based service composition. We think that our research will give us a more principled approach to DSA.

Acknowledgments

Thanks are due to Matthias Wagner, Anthony Tarlano, and Hendrik Berndt 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, 1998.

[4] Czarnecki, K.: *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. Dissertation, TU Ilmenau, 1998.

[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, USE, 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, 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, USE, 2002.

[11] Hürsch, W.L.; Lopes, C.V.: *Separation of Concerns*. College of Computer Science, Northeastern University, Boston, February 1995.

[12] 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, USE, October 1997.

[13] Kay, A.: *Is "Software Engineering" an Oxymoron?* Viewpoints Research Institute, 2002.

[14] 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, 2001.

[15] 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, 1997.

[16] Kiczales, G.; des Rivieres, J.; Bobrow, D.: *The Art of the Metaobject Protocol*. Addison-Wesley, 1991.

[17] 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). November 2001.

[18] Lopes, C. V.: *D: A Language Framework for Distributed Programming*. Dissertation. College of Computer Science, Northeastern University, Boston, 1997.

[19] 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, 1987

[20] Mendhekar, A.; Kiczales, G.; Lamping, J.: *RG: A Case-Study for Aspect-Oriented Programming*. Xerox PARC. Technical Report SPL97-009 P9710044. February 1997.

[21] 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.

[22] Pree W.: *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.

[23] Rivard, F.: *Smalltalk: A Reflective Language*. In: Proceedings of Reflection 1996.