An Approach for Developing Adaptive, Mobile Applications with Separation of Concerns

Nearchos Paspallis and George A. Papadopoulos
Department of Computer Science, University of Cyprus
75 Kallipoleos Str., P.O. Box 20537, CY-1678 Nicosia, Cyprus
{nearchos, george}@cs.ucy.ac.cy

Abstract

Modern mobile computing paradigms have set new challenges for the development of distributed mobile applications and services. Because of the variability which characterizes the context of such environments, it is important that mobile applications are developed so that they can dynamically adapt their extra-functional behavior, in order to optimize the experience perceived by their users. This paper proposes an approach for developing adaptive, mobile applications. It is argued that this approach eases the development effort by clearly separating the work required for the development of the application logic from that required for enabling its adaptive behavior. It is argued that in addition to mitigating the development complexity, this approach also enables a new generation of distributed applications. The novelty in the latter is that the applications can dynamically and collaboratively adapt in an ad-hoc manner to improve the quality of the services offered to mobile users.

1. Introduction

Modern mobile computing paradigms have set new challenges for the development of distributed mobile applications and services. Because of their limited resources and the high variability which characterizes their environments, mobile systems are often required to provide adaptive behavior, something which renders their development process significantly more complex and thus much harder.

Mobile computing is typically defined as *the use of distributed systems, comprising a mixed set of static and mobile clients* [13]. More refined and specialized approaches have also been proposed, including the ubiquitous [17], autonomic [6], and proactive [15] computing paradigms. These approaches suggest a new generation of distributed and adaptive mobile systems, as a means for improving the quality of the services delivered to end users.

Naturally, the development of software applications featuring such a sophisticated behavior is not easy. It has been suggested that while researchers have made tremendous progress in almost every aspect of mobile computing, still not enough has been achieved in dealing with the complexity which characterizes their development [6]. Rather, the necessary development environments as well as system and tool support for such scenarios are still an area of ongoing research [11, 12]. Furthermore, this paper argues that the development complexity grows even further as the boundary between cyber-space and real world becomes increasingly obscured. Consequently, new development approaches and tools are needed to mitigate the increased complexity.

In this respect, this paper proposes an approach aiming at taming the inherent development complexity by separating the two main concerns: implementing the functional requirements of an application and enabling its adaptive behavior. Furthermore, it is argued that this approach can optimize the utilization of distributed devices and resources available to a user, by enabling synergistic collaboration among them. The approach is based on a custom component framework, which is also presented.

The rest of this paper is organized as follows. Section 2 discusses adaptive systems both in general, and focusing on *how* and *where* adaptivity can be applied. Next, section 3 describes a new approach for developing adaptive applications with separation of concerns. It starts with the description of a custom component framework which is designed to enable the development approach, and proceeds to the description of the two development phases. This is followed by section 4 which includes a case study example which illustrates the development process. Finally, section 5 provides conclusions and pointers to future work.

2. Adaptive systems

This section studies different aspects of the services offered by mobile, distributed systems. It first covers how users perceive the interaction with a service as the result of both its functional and non-functional properties, and second, it discusses *how* and *where* adaptivity is enabled.

2.1. Separation of concerns

In this paper, it is assumed that users experience the service as the interactivity with the corresponding service provider. For example, a worker uses her PDA to access information on her dynamically updated schedule. In this case the provided service requires the utilization of a distributed system, where her PDA acts as the service client and a corporate computer acts as the service provider. In addition to the actual service, the user also experiences its extra-functional behavior. For example, a user can perceive the richness of the data (e.g. whether high-quality or low-quality images are attached to her assignments), or the network latency and bandwidth (e.g. how long it took for the PDA to get synchronized with the server).

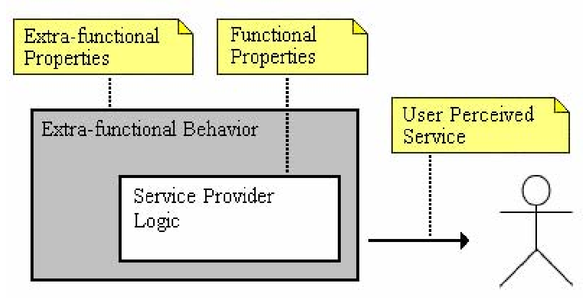

Figure 1: The user's perception of service.

Whether the effectiveness of a service is measured by the existence of some perceived results or the lack of such, as per the ubiquitous computing paradigm, a service can be typically analyzed into two parts. The first part refers to the *functional requirements* of the service, or simply delivering what the service was originally designed for. In the previous example, the functional requirements include the service which allows the user to dynamically access and update her schedule.

As it has been already argued though, real world services are also characterized by what is perceived by users as *extra-functional behavior*. This behavior is generally unpredictable, as it is affected by numerous exogenous factors such as the user occupation, the user location, the network availability, etc. For example, the quality of the attached images and the network latency/bandwidth are examples of how the user perceives the service's extra-functional behavior.

As shown in Figure 1, the user perceives the service as the result of both the application logic functioning, as well as its extra-functional behavior. The main goal of this paper is to propose a software development approach which enables the separation of the two concerns, i.e. developing the application logic and defining its adaptive behavior. With such an approach, the developers would be enabled to concentrate on the functional requirements of their project, rather than mixing the two concerns in the same phase. The adaptive behavior would then be an additional aspect of the application, which could be independently embedded, by exploiting reusable adaptive techniques.

2.2. Enabling adaptations

In [11] it is argued that there are generally two approaches for realizing dynamic adaptations: *parameter* and *compositional* adaptation. While parameters can be used to instruct applications into using different existing strategies, compositional adaptations have an additional benefit: they allow newly introduced components and algorithms which can address concerns unforeseen during development.

This paper introduces separation of concerns in the development methodology of adaptive software. This approach requires and uses reflection and component-orientation. Computational reflection allows a system to reason and possibly alter its own behavior, and component-orientation allows independent deployment and composition by third parties [14].

While compositional adaptation is restricted to the middleware and application layers, parameter tuning also applies to lower layers, such as the operating system, protocols, and the hardware [11]. For example, at the protocol level, the TCP dynamically adjusts its *window* to avoid or recover from network congestion. Also, at the hardware layer, adaptations could target ergonomics (e.g. adjust the display brightness), power management (e.g. turn idle network adapters off), etc.

An adaptive system can be abstracted by a number of layers. The hardware layer, which includes all hardware devices, is right below the operating system and protocols layer. These layers have the common characteristic that any changes in them affect the whole device. For example, if the display brightness or the processor speed is adjusted, all applications using them are affected. Similarly, changes at the operating system layer also affect the whole device. For example, the Windows Mobile operating system allows the adjustment of the storage versus the program memory balance. Clearly, any change in this balance affects the device, and consequently all applications running on it.

On the other hand, changes performed at the layers of components and component-based applications have a more limited scope. For example, it is possible to replace a component implementation, or adjust one of its parameters, without causing any direct effect to the applications that are not using it. At the application layer, adaptations are typically achieved with dynamic reconfiguration (classified as changes to the software implementation, composition, or distribution [5]).

Finally, besides being limited to these layers, adaptations can also extend beyond the boundaries of a single hosting device. This type of adaptations, which are quoted as distribution adaptations, is of particular interest to this paper. It is argued that users can experience great enhancements in the quality they perceive in their software services if the used devices are capable of synergistically sharing services and components, thus better utilizing the available resources.

3. Development approach

This section proposes an approach which can be used for the development of adaptive applications, with separation of concerns. The approach consists of two phases: implementation of the application logic, and definition of its adaptive behavior.

The proposed approach builds on component-orientation, where components are defined as binary units of composition, with contractually specified interfaces and explicit context dependencies only [14]. While components implement services, and they can be composed and deployed (even across devices), services are merely points of component interaction.

3.1. Component framework

The presented component framework was designed to enable the dynamic reconfiguration of component-based applications. It uses Java annotations [1], and it features reflection (i.e. dynamic access to information describing the state, structure and behavior of itself, as well as of its hosted components). The latter is achieved with the use of specific metadata, which are associated with the components in a way which allows dynamic reasoning on their state and use at runtime.

3.1.1. Components and variation points

A prototype component framework has been implemented, using the Java language [8]. The components are defined as *classes*, annotated with both required and optional metadata. Furthermore, the framework implements standardized component containers [3], providing runtime support for dynamic adaptations and lifecycle management. Similarly to the majority of industrial component models (e.g. CCM, COM and EJB), the components are considered as similar to object-oriented classes in the sense that they are instantiated and their instances can be stateful.

Supporting dynamic compositional adaptations (i.e. through dynamic reconfigurations) is a major research area itself. Kramer and Magee have detected a number of important issues for dynamic reconfigurations, most notably the requirement for quiescence [9, 10]. This paper does not aim at providing general solutions to these issues but rather reuses existing ones, such as the configurator pattern [7] which is implemented by the prototype component framework.

The component framework defines a set of metadata which are required to enable dynamic adaptations. These metadata are defined inline with the code using annotations. The attached metadata define information such as a unique identifier, a list of roles they implement and a list of roles they export.

```
@ComponentAnnotation(
  componentID = "cy.ac.ucy.test.AudioProcessor",
  description = “Transforms text to speech”,
  roles = { cy.ac.ucy.test.AudioProcessorRole },
  exports = { cy.ac.ucy.test.AudioProcessorRole }
)
public class AudioProcessor
   extends UnicastRemoteObject
   implements Component, AudioProcessorRole
```

Figure 2: Component annotation example.

In the example depicted in Figure 2, a single class is used to implement a component. The class implements the *Component* and the *AudioProcessorRole* interfaces. The first interface specifies a set of methods, which are used to support the component’s lifecycle control. The latter specifies which Java interfaces are implemented by the component.

Each component implementation is also annotated with information such as a unique identification, an optional description, a list of implemented roles and a list of exported roles (which must be a subset of the implemented roles). Consistency checks ensure that the specified metadata are valid during the component load time. For example, a consistency check verifies that the implemented roles also appear in the class signature as implemented interfaces.

The roles are the main abstraction artifacts, used to enable dynamic application composition. They offer a black box-abstraction, where the offered services are explicitly specified, unlike the internal mechanisms which implement them. In the presented framework, the roles are defined in the form of Java Interfaces, where the specified methods define the offered services. Contrary to the offered roles, the required ones are implicitly defined in the actual components, and they depend on the implementation code rather than the offered roles. With the use of these artifacts, the composition of complex components is broken down to the use of other elementary (or composite) ones, in a recursive manner.

One of the main features of roles is that they can be exported. In particular, certain roles can be shared with applications in other remote devices. As it will be shown, this is achieved by publishing the exported role with a registry, along with its metadata describing the service available by it. Then, other devices can access the registry, and query the set of available services.

The component framework also provides support for specifying binding points. Those are the points where the components connect to each other (in this case they are implemented as RPCs). When a service offered by another component is required, a component specifies and annotates a field of the corresponding type, without assigning a value to it. The following figure shows an example of this:

```
@VariationPointAnnotation(
  variationID = "MyAudioProcessor",
  description = "Requires text to speech support",
  mustBeLocal = false
)
private AudioProcessorRole audioProcessor;
```

Figure 3: Variation point annotation example.

As shown in Figure 3, a binding point can be specified by simply annotating its corresponding field. In this example, the service of an *AudioProcessorRole* is required. For this reason, an un-initialized field of the same type is defined and annotated. The specified metadata include an identification which must be unique within the component implementation, and an optional description. Furthermore, the last attribute specifies whether a non-local component (i.e. hosted on a different device) can be used.

When a component is first loaded, the framework extracts the attached (annotated) metadata and updates the registry with the corresponding information. For example, the lists of available and exported roles are updated. Also, the framework extracts the metadata for all the variation points defined by the component.

Finally, a component is considered to be valid [2] when all its variation points are assigned (i.e. a suitable service provider is available for each variation point either locally or remotely). The components are evaluated recursively, where basic components (i.e. those without dependencies) provide the base cases.

3.1.2. Dynamic composition

The annotation-based mechanisms that were presented in the previous section are used for specifying both the offered and the required roles of components. These roles are then used to facilitate the dynamic composition of applications.

The framework achieves dynamic composition by dynamically adding and removing bindings between components, thus enabling the dynamic configuration and reconfiguration of component-based applications. In this manner, the framework acts as a broker, managing the available and the required roles. Different composition plans are formed by matching required services to offered ones. The actual binding of the components is achieved with the use of reflection, which is a standard feature of Java.

With the use of the annotation mechanisms, the components expose both their required and their offered roles. Using these metadata, the components can be connected to each other to form a composition. Furthermore, hierarchical composition is achieved by implementing the external view of a component through another composition.

One of the main advantages of this framework is that it allows the dynamic planning of compositions, as opposed to frameworks which require a predefined set of possible adaptations. While this paper focuses on compositional adaptation, further adaptivity (e.g. at the hardware layer) is also possible (i.e. parameter tuning).

3.2. Developing adaptive applications

In order to construct adaptive applications, the developers specify *how* an application should be composed, and *when*. The first part is achieved with the construction of components with the use of roles and variation points. The latter also requires a mechanism to reason on the context and to select variations.

Naturally, these two requirements separate the development phase in two parts: developing the application logic and defining the adaptive behavior. An apparent advantage of this approach is that the same components can be reused for the development of additional, adaptive applications (naturally inherited from the component-oriented approach). Furthermore, the same adaptation strategies can be reused in the context of different applications. For example, a strategy which monitors the network requirements of an application, as a function of its components, can be reused for different applications as well.

Finally, because of the high variability which characterizes mobile environments, it is important that the adaptations can be decided and implemented in a quick and efficient manner (i.e. to cope with frequent and unpredicted disconnections). The following paragraphs describe the two required phases.

3.2.1. Defining the application logic

Initially, the developers should specify the core logic of their applications (which is referred to in the literature as application or business logic as well). This is achieved by defining a set of component roles, which describe how the application can be composed.

First, a basic root component is defined. Typically, the root component is fixed and it does not offer any roles (i.e. services). All roles required by it are then specified in the form of variation points which the framework automatically tries to validate. The rest of the application structure is defined in a recursive manner, by the definition of components offering and requiring services through variation points.

It is assumed that the components offering a role are equivalent in the sense that they can be used interchangeably without hurting the consistency of the application. While the reuse of components featuring the same roles is an important and active research topic, it falls beyond the scope of this paper. It is assumed rather, that the components are developed by the same group and role identification is merely used for allowing automated composition planning (as opposed to enabling universal support with ontologies and OCL technologies). Also, it is currently assumed that all possible compositions are valid, i.e. there are no inter-dependencies among components, which cannot be modeled through the presented, recursive approach.

Naturally, multiple components offering the same role can be mapped to the same variation point. The number of valid compositions can be recursively computed using a function f which is defined as $f(c) = g(r_1) + g(r_2) + \dots + g(r_n)$, where $r_1, r_2, \dots, r_n$ are the roles corresponding to the variation points of the component c. The base case of the recursion is $f(c) = 1$, and applies to the case where a component c defines no variation points. Similarly, function g is defined as $g(r) = f(c_1) + f(c_2) + \dots + f(c_m)$, where c_1, c_2, and c_m correspond to the component implementations which offer role r, either locally or remotely (if the variation point allows remote bindings). The total number of possible compositions can then be computed by $f(c_R)$, where c_R is the unique root component. This method assumes no inter-dependencies among components (i.e. all compositions are possible); otherwise, the computed number corresponds to an *upper bound* rather than an exact number.

3.2.2. Enabling the adaptive behavior

So far it has been illustrated how the application is defined as a set of self-described components, and how they can be composed to form a valid application. Nevertheless, the application still lacks a mechanism for reasoning on the context to choose an adaptation.

There are a number of policies which can be used to enable such a behavior. These policies are categorized based on their strategy into *action-based*, *goal-based*, and *utility-based* adaptations [16]. This paper assumes a utility-based approach, where each composition is evaluated with the use of utility functions. The chosen composition is the one which maximizes the utility.

In order to be able to reason about the state of the system and decide on a suitable adaptation, the system must be provided with information describing its context, and the properties of its components.

The context is defined as "any information that can be used to characterize the situation of an entity, where an entity is a person, place, or object that is considered relevant to the interaction between the user and the application" [4]. The context data are used as arguments in the utility functions, which are designed to reason on the suitability of a composition for a given point in the context space. While a context managing mechanism is necessary, the description of one is beyond the scope of this paper. Rather, it is assumed that a suitable mechanism is reused, allowing the framework to register for events concerning a predefined set of context attributes.

In addition to the context, the decision-making mechanism also requires to reason on the components and their properties. The annotated properties of components describe how the components behave in certain contexts. For example, a component can be annotated as "low in processing demands". Similarly, a different component can be defined as "high in battery demands". The exact units and the semantics of a property depend on the component developers. For example, if a component is annotated with a property named "memory-footprint" and a value of "120", it should be understood (by the developers) that the component has a memory footprint of 120Kb.

Thus, in order to specify the adaptive behavior of an application, the developers should first annotate the required components with properties. There is no pre-defined set of properties to be assigned; the developers are rather free to specify any properties which are required by the utility functions.

Once the properties are defined, the next step refers to defining the utility functions. These functions are specified in terms of component properties, and context data. The same utility function gives different results for different compositions. In this way, the system can search for the most suitable composition for a given context state. The evaluation of the utilities is triggered either when a relevant context change occurs, or when the properties of the components change (i.e. a new component is added, or an existing one is deleted). Also, because of the tree-like structure of the compositions, it is possible to evaluate (and reconfigure) only a part of the application (i.e. starting the recursion from a child, rather than the root).

The final requirement is to enable distributed adaptations (i.e. adaptations that involve more than one node). Technically, this is enabled with the use of exported roles, and suitable RPC mechanisms (such as Java RMI). It is assumed that the collaborating devices form a group (e.g. using an existing membership protocol), and that they advertise their exported roles to their group. Adding or removing an offered role from the group registry triggers an evaluation process in the nodes which have at least one application depending on that role. During the evaluation process, the utility of a composition is computed as normally, except when a role available to the group is considered, in which case that is taken into consideration as well.

Of course, before this approach is useful in practice, many issues still need to be resolved. Most notably, suitable protocols are required to coordinate the negotiation between distributed peers. Such protocols should guarantee *fairness* (i.e. ensure the nodes provide and receive services in a fair way), *privacy* and *security* (e.g. prevent malicious users). While this is an open issue requiring further work, it is currently assumed that an appropriate membership protocol is used and that the users are assumed to be fair and trusted.

4. Case study

This section illustrates the proposed development approach with the use of a case study example. This refers to a scenario where a user manages her daily agenda with the use of a PDA, which is capable of interacting with other devices (e.g. through WiFi or Bluetooth connections). Additionally, the application is considered of being capable of interacting with the user either visually or vocally.

4.1. Roles and components

The first step in the development process is the definition of the required roles. An application is always based on a root component. Such a component implements no roles, but rather specifies one or more variation points. In this case, a root component is used, which is identified as *cy.ac.ucy.test.ScheduleManager*.

The main functionality of the application can be abstracted with the requirements to synchronize the user's schedule with the corporate server, and interact with the user to inform her about her schedule. This implies that the root component requires the use of two main roles, namely *cy.ac.ucy.test.ServerAccessRole*, and *cy.ac.ucy.test.UIRole*. The first facilitates read and write access to the server and the second abstracts input and output interaction with the user. The roles are implemented as Java interfaces, where the uniqueness of their signatures is assumed to also imply the semantics of their corresponding services.

The default component (providing the server access role) is a brute-force implementation, using a thin-client proxy to access a remote database. This component is identified as *cy.ac.ucy.test.ThinClient-Proxy* and requires one role (provided by the corporate server). That role allows remote access to the database. The corresponding role is identified with the interface named *cy.ac.ucy.test.DatabaseAccessRole*.

An alternative implementation of this component is one which implements a thick-client proxy, and which is identified as *cy.ac.ucy.test.ThickClientProxy*. In addition to providing read and write access to the server while connected, this component also *caches write operations* on the client side while disconnected (i.e. to allow partial availability during disconnections). Similarly to the thin-client implementation, the only required role is the *cy.ac.ucy.test.DatabaseAccessRole*.

Then, two more possibilities are considered for the implementation of the *cy.ac.ucy.test.UIRole*. First, a default implementation provides visual input and output functionality. For example, the information about upcoming tasks in the user's schedule is displayed in a graphical list, and the user fills the information concerning the completion of her tasks with a visual wizard. This component is identified as *cy.ac.ucy.test.VisualUI*.

An alternative implementation of the same role is also provided to allow audio interaction with the user. To achieve this, the component provides the option of playing audio messages, and accepting vocal input from the user (such as yes or no replies) through voice activation. It is assumed that these components provide sound playing, and voice recognition capabilities. The latter functionality can be actually provided by separate components, which are internally used by the main implementation (i.e. hierarchical composition). This component is identified as *cy.ac.ucy.test.AudioUI*, and has one dependency: a service which transforms text to audio stream, suitable for the player (i.e. implementing the *cy.ac.ucy.test.AudioProcessorRole* role).

The last component refers to the implementation of the *AudioProcessorRole*. This component is simply used to convert text into a byte-stream of synthesized speech. This component is identified as *cy.ac.ucy.test.AudioProcessor* and it has no role dependencies.

4.2. Dynamic composition

At this point, the developer has specified a number of component implementations, along with a set of specifications about the roles they *offer* and the roles they *require*. Given these, the framework can plan a set of valid compositions, as illustrated by Figure 4.

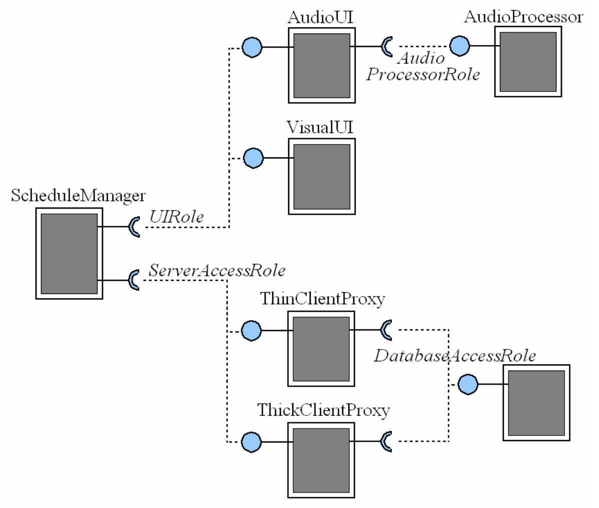

Figure 4: Possible compositions for the schedule manager application.

Assuming that all the components are locally hosted and by using the recursive equations defined in the previous section, it is computed that this example can be configured in four different compositions. Extending this example to allow an additional, remote instance of the *AudioProcessor* role increases this number to six.

As depicted by Figure 2, the *ServerAccessRole* can be offered by two compositions (i.e. a thin and a thick client). These compositions can be combined with two more possibilities, depending on the component used to offer the *UIRole*. Finally, the *AudioProcessorRole* is offered by two possible component implementations (i.e. one local and one remote). This adds two more options (as it can be combined with the two implementations offered for the *ServerAccessRole*) accumulating to a total of six valid compositions.

So far, in this phase the developers have defined the set of components, along with their offered and required roles. These artifacts however, cannot result to an adaptive application until the framework is instructed on *how* and *when* each composition is selected.

4.3. Enabling adaptivity

As it has been argued already, the task of defining how the application is adapted is a different concern, which should be kept as independent as possible from the task of defining the core application logic. In this respect, it is the responsibility of the second phase to define which composition is more suitable for each context in an independent and reusable manner.

Assume that the three context parameters which are relevant to the adaptation decisions in this example are the *network availability*, the *memory usage*, and the *CPU usage*. For simplicity, also assume that these parameters are described by floating-point values in the range of 0 to 1. For example, a 0 value implies no network coverage, minimal memory use and minimal CPU usage, while a 1 value implies excellent network coverage, no free memory and maximum CPU usage.

Given these attributes, the variation points can define utility functions which aim to optimize the use of resources. For example, the *AudioProcessorRole* can define a utility function so that a local implementation is favored when the memory and the CPU usage are low. Otherwise (and if the networking permits) a remote component is preferred. Such a utility function can be defined, for example, as follows: $u(r_A, c) = (1 - memory\_usage) \cdot (1 - CPU\_usage)$ if local, otherwise $u(r_A, c) = network\_availability$. This utility function correlates the utility of a local component to the product of memory and CPU availability, and the utility of a remote component to the availability of the network only. The actual evaluation of the utility functions takes place when relevant context changes occur. At that point, the framework evaluates the utility as a function of the new context and properties, and decides whether a new composition can improve on the existing one.

5. Conclusions and future work

This paper has presented the current state of our work, which aims at defining improved methods for developing adaptive, mobile applications. It is argued that separating the non-functional properties of applications from their business logic, can significantly raise the abstraction level of adaptivity and thus ease the development effort.

A custom component framework has been introduced, aiming to enable the proposed development approach. This framework can be used to ease the development process, and also to enable reusable adaptation strategies. Also, because of the efficiency of this approach, it is argued that it is beneficial for the case of frequently disconnected, mobile environments.

While this approach aims to provide the core on which the development approach is based, a number of improvements are still necessary in order for this approach to become practical for real scenarios. Most notably, suitable negotiation protocols along with the corresponding support infrastructure are required to facilitate a fair and trusted method for sharing resources in the form of exchanged services. Also, the framework should be extended in order to cope with adaptivity to additional layers (i.e. the operating system and the hardware). All these issues, along with an experimentation platform for testing and evaluating different protocols, will be the topic of future work.

6. Acknowledgements

This work was partly funded by the European Union as part of the IST MADAM project (6th Framework Programme, contract number 4169).

7. References

[1] G. Bracha, "JSR 175: A Metadata Facility for the Java(TM) Programming Language", Sun Microsystems, Inc., http://www.jcp.org/en/jsr/detail?id=175, 2002.

[2] H. Cervantes, and R. S. Hall, "A Framework for Constructing Adaptive Component-based Applications: Concepts and Experiences", *7th International Symposium on Component-Based Software Engineering (CBSE 2004)*, Edinburgh, UK, May 25-25, 2004, LNCS 3054, Springer Verlag, pp. 130-137.

[3] E. P. D. Conan, N. Farcet, and M. DeMiguel, "Integration of Non-Functional Properties in Containers", *6th International Workshop on Component-Oriented Programming (WCOP)*, Budapest, Hungary, Jun. 19, 2001.

[4] A. K. Dey, "Understanding and Using Context", *Personal and Ubiquitous Computing,* Volume 5(1), Jan. 2001, pp. 4-7.

[5] C. Hofmeister, "Dynamic Reconfiguration of Distributed Applications", PhD Thesis, Computer Science Department, University of Maryland, College Park, 1993.

[6] P. Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology", IBM Corporation, http://www.research.ibm.com/autonomic/manifesto, 2001.

[7] P. Jain, and D. Schmidt, "Service Configurator, A Pattern for Dynamic Configuration of Services", *3rd USENIX Conference on Object-Oriented Technologies and Systems*, Portland, Oregon, June 16-19, 1997.

[8] J. Gosling, B. Joy, and G. Steele, "The Java Language Specification", Addison-Wesley Professional, 2000.

[9] J. Kramer and J. Magee, "Dynamic Configuration for Distributed Systems", *IEEE Transactions on Software Engineering*, Volume 11(4), 1985, pp. 424-436.

[10] J. Kramer, and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management", *IEEE Transactions on Software Engineering*, Volume 16(11), 1990, pp. 1293-1306.

[11] P. McKinley, S. Sadjadi, E. Kasten, and B. C. Cheng, "Composing Adaptive Software", *IEEE Computer*, Volume 37(7), 2004, pp. 56-64.

[12] B. Noble, "System Support for Mobile, Adaptive Applications", *IEEE Personal Communications*, Volume 7(1), February 2000, pp. 44-49.

[13] M. Satyanarayanan, "Pervasive Computing: Vision and Challenges", *IEEE Personal Communications*, Volume 8(4), August 2001, pp. 10-17.

[14] C. Szyperski, "Component Software: Beyond Object-Oriented Programming", ACM Press/Addison-Wesley Publishing Co., 1998.

[15] D. L. Tennenhouse, "Proactive Computing", *Communications of the ACM*, Volume 43(5), May 2000, pp. 43-50.

[16] W. Walsh, G. Tesauro, J. O. Kephart, R. Das, "Utility Functions in Autonomic Systems", *1st International Conference on Autonomic Computing (ICAC 04)*, New York, NY, May 17-18, 2004, pp. 70-77.

[17] M. Weiser, "The Computer of the 21st Century", *Scientific American* 265(3), September 1991, pp. 94-104.