6 Developing Self-Adaptive Mobile Applications and Services with Separation-of-Concerns

Nearchos Paspallis, Frank Eliassen, Svein Hallsteinsen, and George A. Papadopoulos

6.1 Introduction

Modern trends in mobile computing have inspired a new paradigm for the development of adaptive applications and services. Because of the variability which characterizes the context of such environments, it is important that the software used is developed so that its extrafunctional behavior is adapted at runtime with the aim of dynamically optimizing the end user's experience. This chapter proposes a novel approach for the development of adaptive, mobile applications and services using the separation-of-concerns paradigm (Parnas 1972), as it was studied and designed by the Mobility and Adaptation Enabling Middleware (MADAM) project (MADAM Consortium). The proposed approach specifies a set of steps for developing adaptive applications which can be automatically managed and dynamically adapted by an underlying middleware layer. Most notably, this approach treats the adaptive behavior of the corresponding application or service as a crosscutting concern in which the specification can be separated from the implementation of its functional logic.

In this approach, the applications are assumed to be composed of components and services which are systematically configured (and reconfigured) to offer the highest utility possible to the intended users. In this context, the utility refers to a broad term including both quality-of-service (QoS) metrics and nonquantifiable parameters which generally reflect the extent to which the provided service meets the user needs. The components and services are composed into alternative architectures by means of varying port and service bindings which, as a result, provide different levels of utility to the end users, depending on the context conditions.

In the literature, the separation-of-concerns approach is frequently described as one of three main technologies supporting compositional adaptation (the other two being computational reflection [Maes 1987] and component-based design [Szyperski 1997]). Additionally, the widespread use of middleware technology is typically considered as one of the major catalysts for enabling compositional adaptations (McKinley et al. 2004a). The proposed approach builds on top of these technologies and, furthermore, utilizes a middleware

infrastructure which provides support services. These services include automatic detection and management of context changes, evaluation of the context conditions and user needs, and implementation of runtime service adaptation. The adaptation service relies on the dynamic publication and discovery of components and services, as well as the dynamic discovery of distributed services available in the context of the middleware-managed applications and services.

The developed applications are designed as per the component-oriented paradigm, where a component is understood to provide a service to its clients (i.e., other components and eventually the end users). In addition to this, the components are annotated with a set of metadata describing the relationship between their provided extrafunctional service properties and the required extrafunctional properties from collaborating services. Based on these metadata, and with the support of architectural reflection (Floch et al. 2006), the middleware is capable of dynamically selecting service implementations and applying adaptations as a response to context changes in a way which maximizes the benefit (i.e., utility) to the end users. Ubiquitous services are considered to be reusable and composable entities that can be exploited to improve the utility of a mobile application. In this way, the middleware seamlessly supports configurations based on both components and ubiquitous services.

Besides a middleware system, a development methodology is provided, comprising a set of models and tools which enable systematic and simplified development of adaptive applications and services. The latter is achieved by enabling the developers to concentrate on one aspect at a time, as per the separation-of-concerns paradigm. The relevant aspects in this case include the development of the functional parts of the system, and then the definition of its extrafunctional behavior. The actual aspect of defining the extrafunctional properties of a system is further refined into more granular concerns, such as the application of the where, when, and how adaptations.

The SeCSE model (discussed in chapter 1) provides a loose and flexible definition of services. For the purposes of this chapter, the definition of services is adapted to better reflect the use of services from the perspective of the proposed approach. In particular, software components are considered as entities which offer and require services (also referred to as roles). Dynamic composition of components implies the binding of such services at either a local or a distributed (over-the-network) level. Support is also provided for leveraging common service technologies, such as Web Services, in parallel with the component-based applications. Concerning the taxonomy, presented in chapter 1, this work lies in the layer of service integration, and in particular it proposes development approaches for self-adapted, context-aware applications with emphasis on systems which aim for mobile and pervasive computing environments.

The rest of this chapter is organized as follows. Section 6.2 introduces the concepts of context awareness and adaptivity in the domain of mobile computing. Section 6.3 provides the first contribution of this work, which is the modeling of the context-awareness and

adaptivity aspects as individual concerns. This provides the foundation for an elaborate development methodology and the required supporting tools which are thoroughly described in section 6.4. A case study scenario is then presented in section 6.5, with the purpose of better illustrating the use of the proposed methodology. Section 6.6 discusses related work and compares it with the proposed approach. Finally, section 6.7 concludes the chapter.

6.2 Context-Awareness and Adaptivity for Mobile Computing

Mobile computing is typically defined as "the use of distributed systems, comprising a mixed set of static and mobile clients" (Satyanarayanan 2001). As the popularity of mobile computing constantly increases, the study and adoption of relevant technologies, such as those studied in the ubiquitous (Weiser 1993), autonomic (Horn 2001), and proactive (Tennenhouse 2000) computing paradigms, is a necessity. Altogether, these paradigms suggest the need for a new generation of distributed and adaptive mobile systems, as a means for improving the quality of the services delivered to the end users.

Naturally, the development of software applications featuring such a sophisticated behavior is not easy. It has been suggested that although researchers have made tremendous progress in almost every aspect of mobile computing, major challenges still remain related to the increased complexity which characterizes the development of adaptive software (Horn 2001), and the necessary development methods and environments are still an area of ongoing research (McKinley et al. 2004a). It is also argued that the development complexity grows even further as the boundary between cyberspace and the real world becomes increasingly obscured. These arguments prompt the need for new development approaches and tools, with the goal of containing the development complexity and, consequently, its cost.

Any system which is designed to dynamically modify its implementation and behavior with the aim of optimizing the utility offered to the end users in environments of varying user needs and computing resources must possess two important properties: the ability to sense the environment and the ability to shape it. This is more apparent in applications following the pervasive computing paradigm, where computers are expected to be seamlessly embedded in the fabric of our everyday activities and automatically adapt their behavior (and as a result the environment) as a response to sensed context. In this respect, the following two subsections define and describe the scope of the proposed approach with respect to its context-awareness (environment-sensing) and adaptive behavior (environment-shaping) requirements.

6.2.1 Context Awareness

Context-awareness related mechanisms provide the primary means by which systems sense their environment. Context is commonly 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 a user and an application, including the user and the application themselves'' (Dey 2000, 2001). This definition is followed by another definition that classifies context into three basic categories, based on the type of information it abstracts: user, computing, and environmental context types (Chen and Kotz 2000). The first group includes all types of information describing the physical and mental state of the users, such as whether they are driving, attending a lecture, sleeping, being anxious, or angry. The second group includes the information which can describe the computing infrastructure. This includes the memory and CPU usage of a computing node, the available networks and their characteristics, and the availability of software and hardware components. Finally, the environmental context describes the information which is related to the environment of the entity of interest (typically a user or a set of users), such as the location, the weather, the light and noise conditions, and even the time of the year (e.g., spring or autumn).

In pervasive computing applications, the context information is very important because it provides the primary means for enabling systems to intelligently adapt themselves (and consequently their offered services) to a changing environment. In the original description of ubiquitous computing (Weiser 1993), it was argued that the increasing ratio of computers to users leads to a situation where the users are simply overwhelmed by the demand for interaction. The Aura project (Sousa and Garlan 2002) reiterates this by stating how the increasing numbers of mobile and embedded computers render human attention as the scarcest resource of all, while at the same time resulting in a situation where many devices compete for a share of it. These ideas highlight the need for new channels of communication and interaction between users and computers. For example, the users could trigger appropriate actions in an implicit manner, simply by having the computers sense the context of interest to their domain without requiring explicit interaction with the users.

In this approach, the context information is primarily used with the aim of adapting the extrafunctional behavior of the applications. This is in agreement with the paradigm of developing context-unaware applications, and, as explained in the next section, it facilitates the development of self-adaptive software using the separation-of-concerns paradigm. Explicit use of context information in the functional logic of the applications (such as the use of location information for locating our position in a map tool) is also possible, although not the focus of this work.

6.2.2 Adaptive Behavior

Although context awareness enables sensing the environment, adaptations provide the main mechanisms to shape it. The adaptive behavior of the software refers to its ability to dynamically alter its behavior, using parameter and compositional adaptation. In the first case, a set of variables is defined so that they can be dynamically modified at runtime with

the aim of changing the system behavior. A typical example of this is the Transmission Control Protocol (TCP). In TCP, some parameters can be dynamically adjusted to control the transmission window management as a response to the detected network congestion. By contrast, compositional adaptation enables the structural and algorithmic modification of a system, thus enabling, for example, the introduction of new behavior after deployment, or the correction of software without having to bring the system to a halt and then start it up again. For instance, a video teleconferencing tool could be designed to use alternative encoders and decoders, switching them at runtime without having to halt the system (and very likely terminate the active conferences).

Software adaptation is a well-studied field, and many experts have explored the aspects related to how, when, and where adaptations can be applied. For instance, McKinley et al. (2004a) discuss the most common answers to these questions and also provide an extensive taxonomy of related technologies for each of these questions, in which the where, when, and how are treated as orthogonal dimensions of the taxonomy.

In the approach proposed in this chapter, the adaptations are dynamic, triggered by context changes and enabled by architectural reflection, which allows the complete reconfiguration of service-oriented, component-based applications. Furthermore, the adaptations are enabled by allowing the restructuring of the application, and possibly the replacement of some of its components, or by rebinding to alternative service providers discovered in the environment of the application. Support for dynamic adaptation is an evident need, as mobile environments are naturally characterized by continuous context changes, which in turn require immediate corrective actions. In this way, the overall user experience can be optimized throughout time.

Finally, we examine adaptations at two levels: the application and the middleware. The applications are adapted as a means of changing their composition and/or rebinding to alternative service instances in the environment of the application, which is the primary goal of the system. The middleware itself can also be adaptive, in the sense that it can be reconfigured when necessary. For example, specific context sensors can be dynamically added to or removed from the middleware, depending on the runtime needs of the deployed applications for context information. Furthermore, protocols such as SOAP and RMI, and even proprietary protocols, can be dynamically used in order to bind to newly and dynamically discovered service instances.

In general, compositional adaptations are restricted to the middleware and the application layers, whereas parameter tuning also extends to the lower layers, such as the operating system, the communication protocols, and the hardware (McKinley et al. 2004a). Typical examples of hardware adaptations include the domains of ergonomics (e.g., adjusting the display brightness based on the ambient light conditions) and of power management (e.g., switching off unused or unnecessary network adapters to conserve the battery).

6.3 Crosscutting Concerns for Context-Aware, Adaptive Systems

This section studies different aspects of the services offered by mobile and distributed systems. Furthermore, it describes how mobile users perceive the interaction with a service as the combination of both its functional and its extrafunctional behavior (Paspallis and Papadopoulos 2006). This is followed by a discussion on how, when, and where the adaptations are applied. The results of this section are used to establish the foundation on which the proposed development approach is built.

We view an application as a system providing a service to an end user. Furthermore, applications may depend on the availability of other services which may have further service dependencies. Required services may be provided by bound components or by remote service instances. For example, consider a worker using a PDA to access information about her dynamically updated agenda. In this case the provided service could be implemented as a distributed system, where a local component on the PDA accesses a remote service on a corporate server to fetch task assignments. In addition to the functional properties of the service offered by the application, the user perceives the properties which characterize its extrafunctional behavior. For example, the user perceives the richness of the data (for instance, whether high- or low-resolution images are attached to her task assignments) and the service's responsiveness (for instance, how long it took for the PDA to get synchronized with the server, which is a function of the network latency and bandwidth).

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 (Weiser 1993), a service can typically be analyzed into two parts. The first refers to the functional logic of the service, or simply delivering what the service was originally designed for. This behavior is also commonly referred to as the business logic of the application or service. In this example, the functional requirements include the functioning of the agenda application, which allows the user to dynamically access and update her task assignments.

As it has been already argued, though, real-world applications and services are also characterized by what is perceived by users as their extrafunctional behavior. This behavior is generally highly dynamic and is affected by numerous exogenous factors, such as the occupation or state of the users, the availability of resources such as memory and networks, and environmental properties such as location and light and noise status. In the worker example, the quality of the attached picture and the network latency or bandwidth are examples of how the user's perception is affected by the extrafunctional properties of the service.

As illustrated in figure 6.1, the user perceives the service as the combined result of both the service logic and its extrafunctional behavior. The main contribution of this chapter is the proposal of a software development methodology which enables the separation of the two concerns: developing the application logic and defining its adaptive behavior. With this approach, the application developers would be allowed to concentrate on the func-

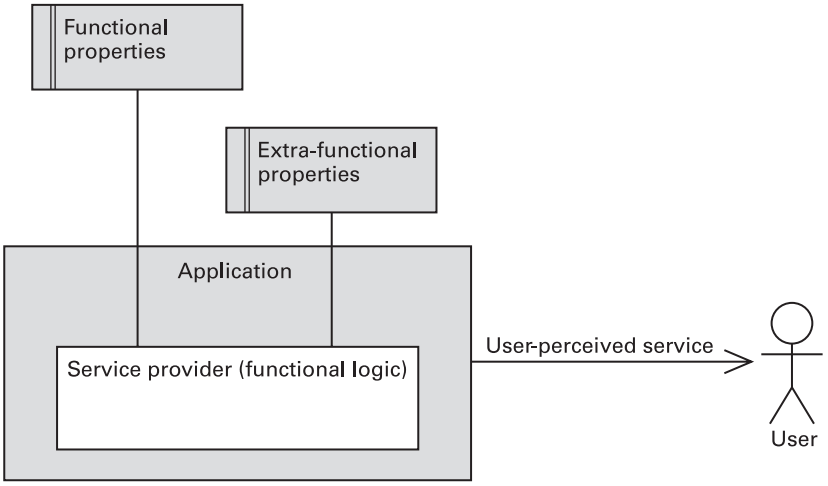

Figure 6.1
The user perceives the service as the combined result of both the application logic and its extra-functional behavior

tional requirements of their project only, rather than mixing the two concerns in the same development phase. The specification of the adaptive behavior would then be an additional aspect of the application or service, which could be developed independently and applied in a different layer (such as in the middleware) by the exploitation of reusable adaptation techniques. A detailed description of how this is achieved is in section 6.4.

6.3.1 Where, When, and How the Adaptations Are Enabled

In this chapter, ''adaptations'' refers to the runtime adjustment of the extrafunctional properties of an application or service by compositional or parameter adaptation, with the purpose of optimizing the utility delivered to the end users. Thus, it is important to study where, when, and how the different adaptations are applied.

Where Adaptations Are Enabled Compositional adaptations are typically applied at either of the middleware layers or at the application (or service) itself. In McKinley et al. (2004a), it is argued that adaptations can be applied in any of the four possible middleware layers, ranked based on the type of services they offer: domain-specific middleware services, common middleware services, distribution middleware, and host-infrastructure middleware. Nevertheless, adaptations (mostly in the form of parameter adaptations) can also be applied in additional layers of a typical computing node. For example, modern operating systems employ several adaptation techniques, mostly to accommodate power efficiency (for example, in the form of spinning a hard disk down, or adjusting the display

brightness when a laptop is unplugged from the power outlet). Also, additional hardware devices can be adapted, usually as a result of an operating system module (for instance, some network cards can be switched off when not needed, whereas others adjust their transmission power based on their proximity to the base station).

When Adaptations Are Enabled Having detected where to apply the adaptations, the next step is to detect the events that trigger them. In the literature, adaptations are usually classified based on the phase of the system's lifetime: development time, compile (or link) time, load time, and runtime (McKinley et al. 2004a). Clearly, the first three imply static compositions, and runtime adaptations imply dynamic ones. The same classification is also used in Aspect-Oriented Software Development (AOSD) (Kiczales et al. 1997), where different aspects of a software system can be weaved during any of these phases. At development time, the adaptations are apparently hardwired into the code, and thus provide limited flexibility. At compile (or link) time, the main enabling technology involves customization of the components, which also provides limited flexibility. Finally, the load time adaptation is typically provided by editing configuration files before loading the software, which implies a slightly more flexible form of adaptivity. In the case of mobile and pervasive computing environments, however, the context changes rapidly, and consequently adaptations are primarily required to be applied at runtime. In the case of context-aware and self-adaptive applications, the decision of when the adaptations should take place is primarily a function on a set of predefined context data types. For example, an adaptation can be triggered by the level of remaining battery power, or by the availability of wireless networking, or, more commonly, by a combination of changes in several context types.

How Adaptations Are Enabled The last step in enabling adaptations concerns the decision on how the adaptations should be enabled. Although many techniques have been proposed, all of them are based on a fundamental approach: the creation of a level of indirection in the interaction between software entities (McKinley et al. 2004a). A large number of enabling technologies is described and evaluated in McKinley et al. (2004a), such as using specific software design patterns (Gamma et al. 1995), Aspects (Kiczales et al. 1997), reflection (both programmatic and architectural), and middleware-based adaptations. In the case of self-adaptive systems, the how question also applies to the approach used to take the decision on which adaptation must be selected. Actually, one of the promises of autonomic computing (Horn 2001) is that the composition of adaptive systems will be controlled by autonomous and completely automated software components. The current state of the art includes three types of adaptation approaches: action-based, goal-based, and utility-function-based (Walsh et al. 2004). Action policies dictate the adaptation action that the system should take whenever a context change occurs (condition for action). Action policies require policy makers to be familiar with the low-level details of the system such that the action policies cover the complete context and application state

space, something that in practice is very challenging (especially as it requires handling of overlaps and conflicts). This is also incompatible with the long-term goal of elevating human administrators to a higher level of behavioral specification, as in our approach. Goal-based approaches define higher levels of behavioral specifications which set objectives (goals) for the state of the system, such as an upper bound for response time, while at the same time leaving the system to autonomously determine the actions required to achieve them. Conflicts arise when the system cannot satisfy all the goals simultaneously (e.g., which goals should be dropped?) or when multiple adaptation alternatives satisfy the goals (e.g., which one to select?). Finally, utility-function-based approaches assign values (utilities) to adaptation alternatives and provide even higher levels of abstraction by enabling on-the-fly determination of the optimal adaptation alternative (typically the one with the highest utility).

6.3.2 Developing Adaptive Applications with Separation-of-Concerns

This section proposes a development approach which consists of two steps, handling each of the two crosscutting concerns. In the first step, the developers break down the functionality of their applications or services into an abstract composition of roles. A role is the abstraction of a part of the application in terms of services provided and dependencies on services provided by other parts, or by the environment. In this phase, the developers need not worry about the potential adaptive behavior of their applications; rather, they simply concentrate on designing the basic components and services required for providing the main functionality (i.e., business logic). In the next phase the developers specify the adaptive behavior of their software. They do so by specifying the compositional and parameter-based reconfigurations which can be dynamically applied as a means of optimizing the utility as that is perceived by the end users. These adaptations are further refined into the more elementary steps of defining where, when, and how the adaptations take place.

The development of the basic functionality logic is the first and primary crosscutting concern to be handled when developing adaptive applications and services. In this respect, our methodology simply implies that the developers use the component-oriented paradigm for architecting and designing their applications and services, as is illustrated in figure 6.2 (part A). In this phase each component is specified by the role it has in the architecture, where a role refers to a service type (i.e., the type of service this role must provide in the composition).

Once the functional design phase is completed, the developers proceed to enable the adaptive behavior of their system by locating the variation points where adaptations are to be enabled. Potential variation points include the roles defined in the previous phase. These variation points are selected so that the users benefit from the resulting variability by being able to optimally exploit different functionality profiles, depending on the changing context. For example, if the service involves video streaming, then an apparent variation point corresponds to the selection of implementation for the compression code

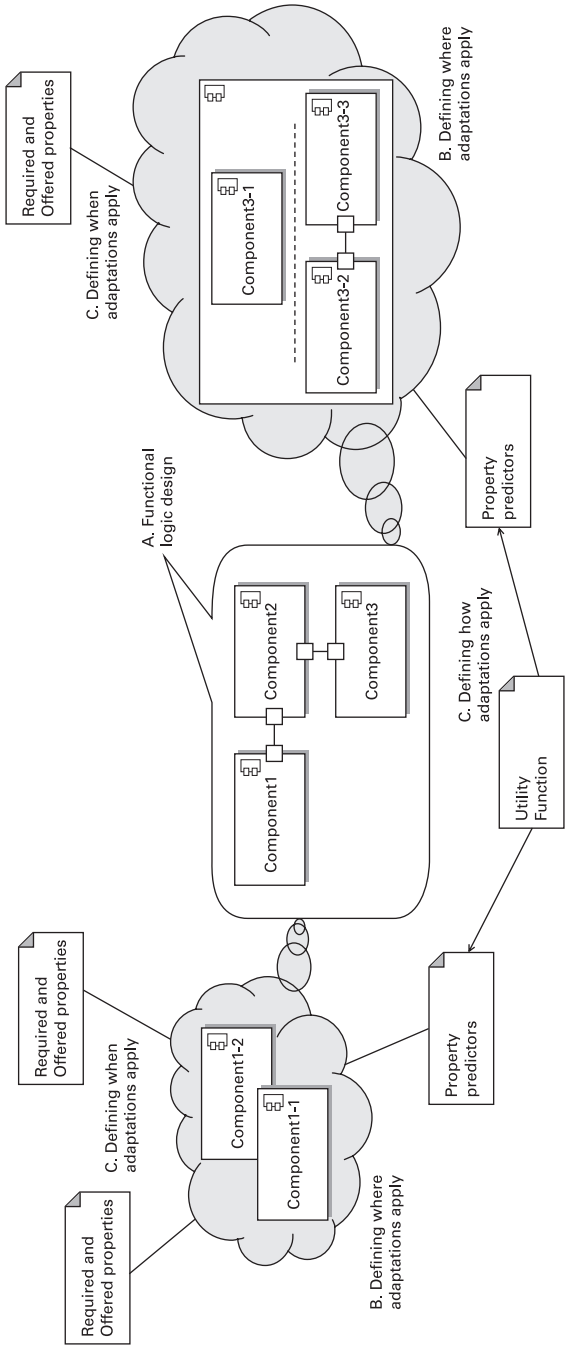

Figure 6.2
Developing with Separation-of-Concerns: specifying where, when and how adaptations take place

component to fill the corresponding role. This selection is naturally based on relevant context data, such as the networking conditions and the user status. Assuming that the users might be required to interact with the developed service in situations which require hands-free operation, the original design might then be extended with additional implementation alternatives to provide the GUI role in order to accommodate such a possibility. Consequently, the variation points might involve different implementation alternatives of existing roles only, or they might involve the introduction of additional composition alternatives which were not present in the original design.

This is illustrated in figure 6.2, where the transition from A to B indicates the specification of the variation points in the basic architecture (functional logic) of the basic application. For instance, in this example two implementation alternatives are specified for component 3: one atomic and one composite. At the same time, the role component 1-1 also has two implementation alternatives which are both atomic components.

In the MADAM approach, the variability is defined by means of three modeling artifacts: blueprint, composite, and service instance plans. The first one enables the use of atomic component realizations as building blocks for the applications. The composite plans are the main modeling artifact for enabling variation at the component level. Finally, the service instance plan enables the use of services for providing specific roles (i.e., subsets of the required functionality).

Following the determination of the application's variability, the developers proceed by specifying when the adaptations take place. This is accomplished by specifying the context and resource dependencies that affect the extrafunctional behavior of the application. Changes in these context and resource values are the triggers for adaptation. This knowledge of dependencies is encoded as metadata associated to individual components. The metadata can be used to characterize the offered extrafunctional properties of the service implemented by each component, as well as the corresponding extrafunctional properties of required collaborating services. They can, for example, be used to characterize some expected properties of the running environment (such as networking bandwidth of at least 10Kbps). Because sometimes the properties cannot be expressed by constant values, additional support for property predictors is also made available, which enables the expression of a property as a function of other properties and possibly context values. For instance, the responsiveness of a service offered by a component can be described as a function of the latency and the bandwidth of the network connection which is used to bind it to the remote service provider. This is illustrated in figure 6.2 as well, where the transition from B to C is achieved with the annotation of the different variation options with appropriate property metadata.

Finally, once the functional design and the specification of the variation points, context dependencies, and extrafunctional properties are completed, the developers need to describe how their applications and services must be adapted once an adaptation is triggered. As already discussed in section 6.3.1, autonomic systems typically employ action-based,

goal-based, and utility-function-based approaches. In this approach, we opt for the last one, where the property predictors and the utility functions are designed so that they reason on the context situation by means of computing the utility of potential component compositions as numerical values. The task of planning the set of feasible compositions and computing the utility of each one is left to the composer, which in our approach is the adaptation manager described in document D2.3 of MADAM Consortium. Though the utility function provided by the developer determines the optimal configuration for a given context, the adaptation manager determines the detailed reconfiguration steps needed to bring the application into the selected configuration. Figure 6.2 (transition from C to D) illustrates the addition of the metadata required for the specification of the utility functions and the property predictors.

6.4 Development Methodology and Supporting Tools

This section describes a development methodology which is based on the approach outlined above and which addresses the different concerns in separate development steps. One of the main contributions of this proposal is a conceptual framework and modeling notation for adaptive applications meant to operate in mobile environments. The methodology applies this notation to gradually analyze and model the application in steps, tackling a different concern at each step. In the first step, the initial component architecture is developed based primarily on the functional requirements. Then, in the following steps, the three adaptation concerns (where, when, and how) are addressed. This section describes these steps by covering both the analysis and the modeling approaches.

6.4.1 Defining the Initial Architecture

Just as in traditional software engineering, the first step when designing a new adaptive system is the analysis of the problem domain, aiming to elaborate the understanding of the problem to be solved and to define the core functionality of the system and its interaction with its environment. This is followed by the derivation of the initial component architecture. The recommended notation for this step is UML composite structures, which allows defining the system and its environment as a set of entities along with the roles they should implement. In this case, the entities refer to anything that can affect the interaction of the user with the application, including the application and the user themselves (Dey 2000). The roles are simply used to denote the interaction between entities, and they can correspond to the binding between ports of interacting software components, or artificial ports representing the interaction between a user and an application (i.e., a user interface component).

For example, if the application requires vocal interaction with the user, then a component must be defined to transform text-based messages to speech. Additionally, a basic architecture for connecting these components must be provided so that the application is

operational. This architecture specifies the structure of the application as a realization of a composite component, which itself can be recursively decomposed into further, possibly composite, components as well.

As the study of general techniques for the analysis and the design of the functional part of component-based applications is beyond the scope of this chapter, it is assumed that the developers exploit existing approaches for identifying their applications' functional and QoS requirements. This process is sufficient for the completion of the functional specification of the application, which is the first crosscutting concern of the proposed methodology.

6.4.2 Defining Where the Adaptations Take Place

The next step focuses on the adaptation concern: what requirements and constraints are likely to vary and where we have to insert variation points in the implementations in order to be able to adapt it. Typical factors that vary for mobile applications include the computing infrastructure, the environmental conditions, and the user. The anticipated variation is modeled by annotating the appropriate entities of the model developed in the previous step with varying properties, expressed as a property name and an associated set of possible values. In effect, this can be considered as an interaction-centered approach, as the two principal actors of the interaction are considered first (the user and the computing infrastructure) and the environmental context (i.e., surrounding the interaction) is considered next.

Having identified the adaptations, the next step includes the specification of a set of variation points to enable the adaptations of the application in order to suit the current requirements and constraints. In this step, the developers consider the implementation and realization of the components which are required to fulfill the roles defined in the initial architecture. Furthermore, they recursively abstract components by high-level roles, which in practice can be faced as generic (possibly composite) components or services exporting the same functionality (through the same role) and implementing the exact same logic. This abstraction enables the developers to provide different realizations for each role: either a component type, which can be realized by either a composite or an atomic component, or a service description type, which realizes an equivalent role. The atomic components provide a realization which cannot be decomposed any further, whereas the composite components can be further refined into additional component frameworks of more elementary components, in a recursive manner. Finally, the service description provides appropriate metadata describing alternative service instances that can be used to provide the corresponding role. The described variability model builds on original results from Geihs et al. (2006) and is depicted in figure 6.3.

In addition to local compositions, the composite component offers the possibility for defining distributed deployments for the application. This can be done by specifying where each component type of a composite composition is deployed (i.e., on the mobile device or

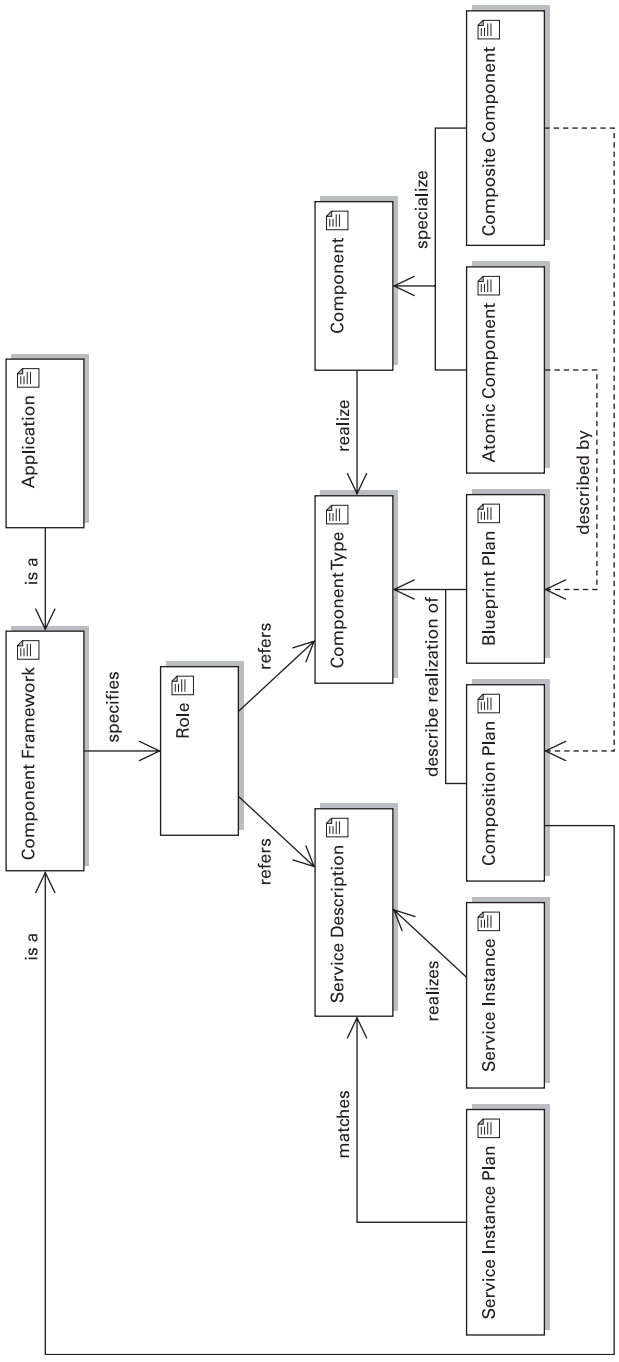

Figure 6.3
Variability model

on a server). As discussed in section 6.4.5, this approach supports a limited form of distribution in which some components are allowed to be deployed remotely, with the aim of optimizing the resource usage on the mobile device as well as the quality of service offered to the end user. In this respect, the developers should specify in this phase which components are subject to distribution, by modeling them with the corresponding composite composition. Further distribution is also achieved through the use of services (e.g., Web Services), but the difference in this case is that the actual implementation and management of the service is beyond the explicit control of the corresponding application.

Naturally, there are dependencies between variation points, in the sense that a choice at one variation point restricts the possible choices at others. To reflect such dependencies, the developers specify a set of variability constraints that determine which variants are feasible and which are not. For example, an application which provides the possibility for selecting alternative atomic realizations at two points can ensure that in no case is a composition chosen in which the two selected atomic components are contradicting (e.g., one instance requiring networking and the other compromising the user utility by avoiding network use to achieve lower power consumption). The details of this mechanism are outside the scope of this chapter; interested readers are referred to document D3.3 of MADAM Consortium.

From a practical point of view, this step results in the definition of the application's variability model, through the provision of a set of possible compositions. As is illustrated in figure 6.3, an application is defined as a component framework which specifies one or more roles. The roles are simple abstractions of a particular service, as it can be provided by a component type or a service, and provides a well-defined interface to enable this abstraction. On the component type side, multiple alternatives for atomic and composite components are enabled through their modeling as blueprint and composition plans, respectively. On the other hand, a role can also be provided by an instance of a service as it is modeled by a service instance plan. Given this variability model, the underlying middleware can dynamically generate the complete set of all possible variations. Furthermore, this model allows the consideration of components and component types which are dynamically added (or removed) at runtime, even after the initial application deployment.

Finally, the actual service instance plans can be provided either statically at development time or dynamically at runtime. In the first case, the developers specify a number of predefined service instance plans that correspond to actual, well-defined services (i.e., as they are specified by a URL). In the second case, the service instance plans provide just the specifications of the actual service instances, and an appropriate module in the underlying middleware actively searches for potential providers of the corresponding service. In both cases, the services are abstracted by a role which in practice can also be provided by a component type. This implies that alternative composition plans can be dynamically generated in parallel, so that some of them involve the use of services, and some others do not.

6.4.3 Defining When the Adaptations Take Place

By expressing the variability possibilities for an application, the developers implicitly define a number of possible variants, each one of which is better suited for a particular context. For example, one variant of a video code component can be optimized for power efficiency but provide lower video quality, whereas another can be less power-efficient but provide better video quality. In this way, different variants offer varying levels of QoS (and thus utility) to the end users, and at the same time exhibit different patterns of resource consumption.

In order to support automatic adaptation, we need to model how the various choices at the variation points affect the properties of the system. This is done by defining property predictors associated with the components and the application. The property predictors are functions computing the varying properties of the component to which they are associated.

The analysis on the computing infrastructure includes reasoning on the context requirements (e.g., the selection of an appropriate variant might be affected by whether there is networking available, or whether there is sufficient free memory available). However, as the software components themselves are part of the computing context, the developers should also provide an analysis on the characterization of the components (e.g., the memory footprint of a component is 10kB or the minimum networking requirement for another component is bandwidth of 20Kbps). Additionally, the developers must reason about their applications' dependency on environmental and user properties, such as the ambient light and noise conditions, or the users' need for hands-free operation (e.g., while driving).

In conclusion, the result of this phase includes a set of context properties and related property predictor functions that allow the estimation of the properties of the application variants. This information abstracts offered and needed properties of the entities, whose values can be expressed either as constants or as functions on other properties.

6.4.4 Defining How the Adaptations Are Decided

The last step in the problem analysis includes the determination of how the application adaptations are decided (in other words, which variant should be selected for each possible combination of context conditions). In practice, this is the step where the developers are expected to specify the autonomous, self-adaptive behavior of their applications.

In this approach, a utility-function-based approach is used which is considered to be the most suitable for the proposed development approach and middleware design. In this approach, the developers are expected to specify a utility function that is used to rank the potential variants based on their fitness for a particular context. Ultimately, this utility function is used to select the most beneficial application variant. In practice, the utility functions are generally expressed as polynomials summing the weighted differences between the offered and needed properties of the examined variant. Eventually, a single utility function is associated to one application and is reused when evaluating the possible variants.

Currently, there are not many well-defined and structured methods for specifying such utility functions. Rather, this is an open problem in research areas such as autonomic systems and artificial intelligence. Concerning the approach proposed here, the developers are expected to collect as much information as possible about the desired behavior of the developed application and to use their personal experience and intuition in order to form good utility functions. Once this is complete, the resulting utility functions can be partly validated by testing them with the use of a simulation environment. Such a simulation can, for instance, benchmark the utility of all potential application variants, given a large set of context states.

6.4.5 Dealing with Distribution

An additional crosscutting concern which is automatically handled by the middleware is the support for distribution. Besides distribution at the context level (Paspallis et al. 2007), the proposed approach supports distributed application configurations. This is achieved through the formation of distributed adaptation domains (Alia et al. 2007a). Each adaptation domain corresponds to a collection of computing nodes along with their own resources and the communication networks they are connected to. It is assumed that within an adaptation domain, all computing nodes run an instance of the middleware, or at least a subset of it allowing the formation of distributed configurations. Each domain is associated with exactly one client node and zero or more server nodes. Additionally, the domains are formed dynamically through the use of a discovery protocol where the participating nodes periodically advertise their presence and form loosely coupled group memberships (Paspallis et al. 2007). Unlike clients, servers may be shared, which implies that they can participate in multiple adaptation domains.

The adaptation reasoning is centralized and is always performed under the control of the client node, whereas the adaptation reasoning can be performed at either the client node or the server node to save client node resources. The client node is typically a mobile device carried by the end user. In this case, the client side is granted complete control of the allocated resources on the server nodes (Alia et al. 2007b), and is fully responsible for making the adaptation decisions. Besides its local context information, the client node is provided with access to the available resources of other nodes, and thus it is rendered capable of making centralized decisions which include elements from the wider context of the distributed application. The resulting adaptation can involve the deployment of components on individual, remotely connected nodes, thus resulting in a distributed configuration.

Finally, the applications running inside a domain may depend on services provided outside the domain. These may include both Web Services and shared peripherals. Discovering, selecting, and binding to suitable service instances is also part of the responsibility of the adaptation management component. Naturally, this responsibility extends to replacing or removing the need for services that disappear or otherwise break the service-level agreement (SLA).

The general middleware functionality, as well as the behavior of the system in the case of distributed adaptations, is illustrated in section 6.5, where a case study example is described by providing information about both the development methodology and the deployment process.

6.4.6 Implementation and Testing

Just as in general software engineering, the completion of the problem analysis is followed by the implementation and the testing of the application. This facilitates the process for readying the application for deployment. Although the described approach is not dependent on any specific platform or programming language, it is assumed that an object-oriented language and an underlying virtual machine environment are used. The latter provides support for interfaces, computational reflection, and the creation of software components. In the MADAM project, the middleware and the applications were implemented using the Java language (Arnold et al. 2000) and were evaluated on mobile devices such as Windows Mobile-based iPAQ 6340 PDAs.

In practice, the proposed methodology is not dependent on any specific component technologies, as long as they support architectural reflection and dynamic reconfigurations. For example, in the MADAM prototype implementation, a simple and custom component framework was defined. In this case, the developers were simply expected to define the functional aspects of their applications by developing the required components and the basic architecture. Both the basic component functionality and the application's extrafunctional features are expressed programmatically by reusing and extending custom-made APIs. For instance, a utility function is expressed by implementing a specific interface and by programmatically defining how the utility is computed as a function of other parameters, including the context.

Besides this programmatic approach, a Model Driven Development (MDD)-based methodology was proposed by MADAM Consortium. This approach includes a set of required models which are based on and extend the Unified Modeling Language (UML) 2.0 standard. The provided modeling artifacts enable the software developers to visually design their applications and express their extrafunctional properties. For this purpose, a number of UML extensions (also denoted as UML profiles) are provided, including the Context, the Resource and Deployment, the Property, the Utility, and the Variability profiles. These profiles are used to incrementally model the different aspects of the functional and extrafunctional parts of an application. As per the MDD approach, a complete set of tools is provided, which not only enables the design of the Platform Independent Model (PIM) of the application, but also allows for the automatic generation of Platform Specific Model (PSM) implementations.

As a proof of concept, a tool chain was implemented for generating application code targeting the Java Virtual Machine (JVM) (Lindholm and Yellin 1999). Although these tools produce code which is complete with respect to their adaptation functionality, in

some cases the developers are still expected to fill in some gaps and provide code snippets which cannot be automatically generated. For instance, utility functions and property predictors are generally defined manually. A detailed description of the corresponding MDD-based methodology is presented in document D3.3 of the MADAM Consortium.

Concerning the testing phase, there is currently limited automated support. Most notably, all context parameters can be simulated at runtime, and thus allow a developer to more easily evaluate the behavior of the application. However, a missing part of the testing framework is a suite which could automatically build the set of possible variants while offline, and dynamically evaluate the utility of each of them for different context values. Such a tool would be of great assistance to developers during the specification of the utility functions and the property evaluators, as it would provide instantaneous feedback concerning the adaptive behavior of the designed application. Although not available yet, there are plans for providing such functionality as part of the MADAM project's successor, MUSIC (MUSIC Consortium).

6.5 A Case Study Example

To illustrate the proposed methodology, this section explains the development of an example application following the steps described above. The case study also illustrates how self-adaptation is important to mobile users for retaining both the usefulness and the QoS of the provided service. The purpose of the example application is to assist a satellite antenna installer with aligning the antenna to the appropriate satellite. This task requires the use of hands and eyes to manipulate the antenna equipment, and therefore the installer in some periods prefers an audio-based interface, whereas a traditional interface using the keyboard and display is preferred in other periods.

The application includes signal analysis, which is quite heavy for a handheld device in terms of both memory and processing requirements, and consequently there is a concern about battery lifetime. Also there is a need to coexist with other applications, for example, a chatting program to be able to communicate with the company's headquarters. Therefore the application should provide the possibility to offload some of the computation to a server available over the network. However, the network connection available varies as the user moves about. Some places feature WLAN coverage, and some others feature weak GSM signals only. Furthermore, the precision required for the signal analysis varies during the alignment operation. The initial steps require lower precision, as opposed to the final fine-tuning steps. To further illustrate the adaptation aspect of this example application, let us consider a typical usage scenario.

First the worker is still in the office, using the PDA to prepare for the day, with several onsite visits scheduled. In this situation, the application has the full attention of the user, and consequently the visual interaction mode is preferred, as it is more responsive and more resource efficient (it requires less memory, less CPU use, and no networking, which

results in lower power consumption). Then the worker moves onsite and starts working on the alignment of an antenna, and the application switches to the audio interaction mode to allow the worker to use her hands and eyes to manipulate the antenna. Later in the day, the worker moves to a site where the network is sufficiently fast and cheap but the memory and CPU resources are running low (e.g., because the PDA starts other applications as well). At that point, the application switches to the audio mode, where the speech-to-text component is either hosted by a remote server or the equivalent role is provided by a service provider.

As is illustrated by this scenario, the self-adaptation mechanism has to monitor the context (i.e., the status of the hosting device, the user, and the surrounding environment) and dynamically respond to context changes by selecting and applying the most suitable application variant at any time. To keep things simple, the further elaboration of this scenario considers two variation points only: the selection of a UI modality (i.e., visual or audio) and the choice between high- and low-precision calculations. For the audio UI case we also consider alternative implementations with different quality and resource needs.

6.5.1 Designing the Example's Initial Architecture

In this phase, the developers detect the primary requirements for the development of the application and break it down into a composition of abstract components or roles. The resulting component architecture for the example includes five abstract components, as is illustrated in figure 6.4.

These abstract components are the UI, the Main Logic, the Analyzer, the Satellite Adapter and the Math Processor. The UI component implements the user interface. The Main Logic component provides the basic business logic of the application, which primarily gathers information from the Analyzer and communicates it to the UI in a human-understandable way. The Analyzer is responsible for reading the received satellite signal

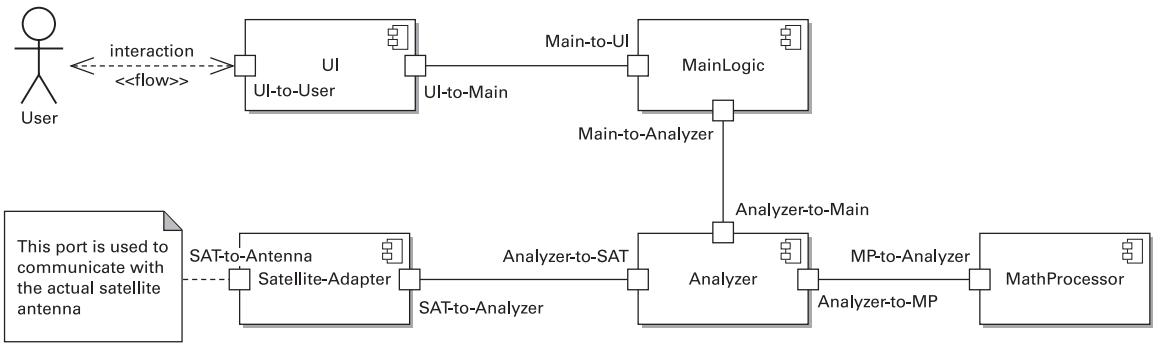

Figure 6.4
Specifying the functional architecture of the case study example

and extracting the information which is required by the user in order to fulfill her assignment. The Analyzer interacts with the Satellite Adapter, which transforms satellite data to an appropriate, computer-usable form, and with the Math Processor, which provides mathematical methods needed for signal analysis. The architecture diagram also specifies how the components are connected in terms of ports and connectors.

Figure 6.5 specifies the relation to the user and the environment. The user is depicted as the corresponding actor using the application, and the Satellite Adapter is shown to interact with the satellite equipment.

6.5.2 Design of Where the Adaptations Take Place

From the description of the requirements for the example application and the scenario, we conclude that the properties relevant to adaptation for this application are audio or visual interaction, the precision of the signal analysis, the need for memory, the need for network capacity, and the power consumption.

Apparently, the selection of where to enable adaptations is driven by the need for adaptation on individual components, as well as on the compositions of the components. For instance, in this case a developer would consult the scenario analysis, which hints at two variation points: the first one involves the selection of an appropriate UI mode, and the second one involves the selection of a realization of the Math Processor type. It is worth noting that although the scenario analysis is the primary input available to the developers, additional input might also be considered, such as whether alternative realizations of relevant component types are available, and whether their properties make them better suited for varying context situations. Additional factors affecting this creative process include considerations on the distribution of components, and whether the distribution of selected component types is desired in certain contexts (or even inevitable in some others).

Though numerous adaptations could be possible in this scenario, for the sake of simplicity it is assumed that the Main Logic, the Analyzer, and the Satellite Adapter are provided by concrete atomic realizations (not shown). The UI, on the other hand, is assumed to be adaptable, and thus abstracted by a composite realization. The Math Processor is also assumed to be adaptable, by providing alternative atomic realizations to fill in the corresponding component type.

The basic variation in the UI consists of the choice between a visual-based and an audio-based interface for the user. In the first case, the user interacts by pressing buttons in the application's display window and by reading the displayed messages. In the case of audio-based interaction, the user makes selections in the application using voice-activation technology, and the application notifies him or her with messages played on the device's speakers. The visual interaction component can be easily implemented by an atomic realization, whereas in the case of audio-based interaction, the implementation is further decomposed into three additional components: the audio controller, the player, and the text-to-speech (TTS), as shown in figure 6.5. Again, for the sake of simplicity, we consider

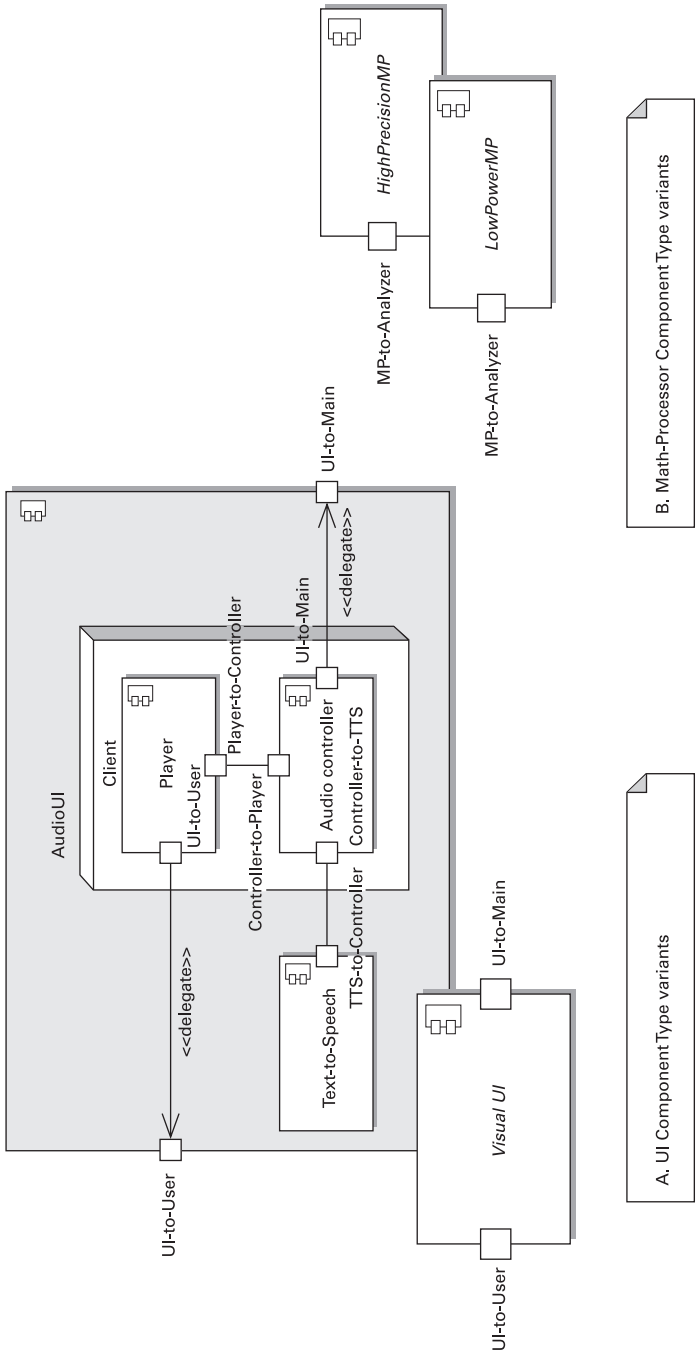

Figure 6.5
Specifying the variability model of the case study example

only the speech output part of the audio UI. The player plays a given audio stream on the device's speakers. The TTS implements speech-synthesis functionality, and the audio controller interacts with the rest of the application and coordinates the TTS and the player.

The player and the audio controller are constrained roles that can be bound only by a component instance deployed on the client device. The TTS role, on the other hand, is an open role that can be filled either by a component instance deployed in the client device or by a remote service providing the same functionality. In our case, we assume that the TTS component takes text as input and returns that text as synthesized speech encoded in the output byte stream. Evidently, the speech synthesis is a very demanding process, requiring significant resources in terms of CPU and memory. For this reason, using a remote service to provide this functionality will in many situations be more suitable.

Figure 6.5 (part B) depicts the two possibilities for realizing the Math Processor component type. In this case, the variability is simply denoted by the availability of the two atomic realizations of the matching component type. Since the selection of the UI component type is orthogonal to the selection of the Math Processor type, the two Math Processor realizations double the number of possible variants to six. Furthermore, assuming that the TTS functionality can also be available as a service, then additional variations are also possible: two for each TTS service provider available. For example, assuming that there is one well-known service type offering the TTS service, then a well-defined TTS proxy can replace the TTS component in the audio UI composition, thus adding an additional composition for the UI component, and two additional compositions to the overall application (corresponding to the two combinations possible with each of the two Math Processor components). Note that in figure 6.5 the visual UI and the two Math Processor components are underlined to indicate their status as atomic realizations rather than composite plans. In the case of the audio UI composition plan, the player, audio controller, and TTS are all abstracted as roles, to indicate that different realizations of each one of them might be plugged in their role.

6.5.3 Design of When Adaptations Take Place

Having specified the variation points, the developers next need to specify when the adaptations are triggered. More specifically, the types of events that can potentially cause the selection of a different variant must be detected. This type of analysis is partly dependent on the previous step, since the developers need to evaluate which types of context changes can potentially trigger the adaptation at any variation point.

With regard to the UI variation point, a developer can detect numerous factors affecting the ultimate selection of interaction with the user. Such factors include the resources of the mobile device (for example, its memory and CPU), the network availability (along with properties such as bandwidth, latency, etc.), and of course the user's needs, such as the need for hands-free operation. Also, the availability of suitable TTS service providers is a significant factor affecting the adaptation decisions. Concerning the second variation

Table 6.1
Resources required by the application and by component realizations

	Memory	CPU	Net bandwidth	Net latency
Minimum resources required by compositions				
Application	UI.m + MP.mem + 100	UI.cpu + MP.cpu + 30	UI.net bandwidth	UI.net latency
Audi UI	PL.mem + AC.mem + TTS.mem	PL.cpu + AC.cpu + TTS.cpu	TTS.net bandwidth	TTS.net bandwidth
Minimum resources required by atomic component realizations				
Visual UI	40	20	0	0
TTS component	80	40	0	0
TTS service	5	5	20	40
Player	30	30		
Audio controller	10	10		
High-precision MP	70	50		
Low-precision MP	50	30		

point, the selection of the Math Processor realization can be influenced by factors such as the available resources (memory and CPU), the user need for short response times, and the user need for precision.

Besides the contextual factors that affect the QoS offered to the end user, an additional concern is the actual requirements of the component realizations with regard to resources. Evidently, before the developers can even start thinking about optimizations through adaptation, the possible variants must be compared against the available resources in order to reason whether they can be realized at all. For example, variations which include component realizations that require networking when that is not available should not be considered at all.

The result of this phase is a matrix with the offered and the needed properties of the involved component types and their corresponding component realizations. Table 6.1 illustrates an example of resource requirements of the individual component types and their realizations. Furthermore, this table depicts the offered utility of the relevant component types (as constants or property predictors), as well as the offered utility of the whole application, as a utility function. For simplicity, it is assumed that all values in this table are in the range of 1 to 100.

As illustrated in this table, both component types specify needed and offered properties as functions on the chosen component realizations. This is compatible with the general approach of both dynamically generating the possible variants (using the variability model) and computing their required and offered properties. Furthermore, the component realizations also specify some of their context dependencies as functions (possibly including conditional expressions) on other context entities, although most of their values are

Table 6.2
Offered properties of compositions and component realizations and utility of the application

	Hands-free	Response	Precision
Properties offered by the component realizations			
Visual UI	0	70	
TTS component	100	50	
TTS service	100	if(ctxt.bandwidth>50) then 70 else 20+ctxt.bandwidth	
High-precision MP		75	80
Low-precision MP		90	60
Properties (utility) offered by abstract component types			
Application	UI.hands-free	$c_1 \cdot$ UI.response + $c_2 \cdot$ MP.response	MP.precision
AudioUI	100	TTS service.response	
Utility function			
Application	utility = $c_H \cdot$ (hands-free=ctxt.hands-free) + $c_R \cdot$ (if (response>ctxt.response) then 1 else (ctxt.response-response)/100) + $c_P \cdot$ (if (precision>ctxt.precision) then 1 else (ctxt.precision-precision)/100)		

specified as constants. In the case of functionally specified properties, the context prefix is used to denote a value read from the context system.

6.5.4 Design of How Adaptations Are Decided

The last step in the specification of the application's adaptive behavior concerns the definition of an appropriate decision mechanism. Although it would be possible to consider any of the three classes of solutions that were discussed in section "Where, When, and How the Adaptations Are Made," in this approach we focus on the use utility functions, which match the concepts of properties and property predictors perfectly, and thus provide an ideal candidate for the proposed methodology.

In this methodology, the users specify only a single utility function which can be interchangeably applied to any possible variant. In table 6.2, the utility function is defined as the weighted sum of a fitness metric for the three main properties needed: the hands-free, the response time, and the precision. This is expressed in the Utility function cell by comparing the required properties (e.g., hands-free) with the provided properties as specified in the context (e.g., ctxt.hands-free). The results of these comparisons are then accumulated and weighted to derive the final utility value which is used to compare the different variants.

For instance, when the hands-free property is evaluated to match (i.e., it is both offered and required), then the first section of the utility function evaluates to 1 and is multiplied by its corresponding weight (i.e., C_H). Furthermore, as also shown in table 6.2, the required properties are computed as constants (e.g., the response property offered by the visual UI is 70), or they are dynamically computed through property predictors (e.g., the response of

the audio UI remote variant is computed as a function of the available bandwidth, and the application's hands-free property is computed as a delegate of the UI type's hands-free property).

The computed values are finally summed up based on their assigned weights. The three weights c_H, c_R, and c_P encode the importance of each of the three properties to the final variant selection. Typically, these weights are directly related to the user preferences. For example, someone might rank support for hands-free higher than that for precision, whereas someone else might value precision as the most desired property.

Though the weights should generally be defined so that adaptations would match the preferences of the most typical users, it should be possible to let the users adjust these values if they need to. In the prototype implementation of MADAM, this was possible through a GUI that enabled users to edit their preferences in the same way they edited the values of the simulated context types. Though more sophisticated techniques could be applied to allow for automated adjustment of these parameters, at this point simplicity was chosen over complexity. In the future, we will endeavor to investigate the application of known algorithms, such as from machine learning (Alpaydin 2004), with the purpose of enabling automated adjustment of the adaptation weights as a self-learning system.

6.6 Related Work

There is a substantial amount of literature on adaptive mobile systems. The Reconfigurable Ubiquitous Networked Embedded Systems (RUNES) middleware (Costa et al. 2005) targets embedded environments and aims at enabling advanced scenarios in which devices leverage off each other and exhibit autonomous and coordinated behavior. Similar to the MADAM middleware, RUNES specifies components which interact with each other exclusively via interfaces (i.e., offered services) and receptacles (i.e., dependencies). Additionally, the RUNES middleware specifies a reconfiguration metamodel based on logical mobility, as described in Zachariadis and Mascolo (2003).

The Odyssey project (Noble 2000; Noble and Satyanarayanan 1999) consists of a set of extensions to the NetBSD operating system which aim at supporting adaptation for a broad set of applications. These applications run on mobile devices but access data on servers. Odyssey supports fast detection and response to resource availability (i.e., agility), but the applications are still expected to independently decide how to adapt to the notified changes.

The Aura project (Sousa and Garlan 2002) primarily targets pervasive applications, and for this reason it introduces auras (which correspond to user tasks) as first-class entities. To this direction, the same project categorizes the techniques which support user mobility into use of mobile devices, remote access, standard applications (ported and installed at multiple locations), and use of standard virtual platforms to enable mobile code to follow the user as needed.

A communication-oriented approach is proposed by LIME (Picco et al. 1999), in which the mobile hosts are assumed to communicate exclusively via transiently shared tuple spaces. The model used offers both spatial and temporal decoupling and allows adaptations through reactive programming (i.e., by supporting the ability to react to events). Although this middleware architecture supports seamless support for distribution, it does not go far with regard to providing support for generic context-aware adaptations.

The Quality of Service Aware Component Architecture (QuA) project investigates how component architectures can preserve the safe-deployment property for QoS-sensitive applications (Amundsen et al. 2004). Similar to the MADAM approach, the QuA project envisages platform-managed QoS, where the platform is able to reason about how end-to-end QoS depends on the quality of component services.

In another work (Lundesgaard et al. 2006), a service model is proposed which classifies services in three levels: service, subservice, and atomic service. This classification is similar to what is used in service-oriented computing (SOC), where applications can be seen in terms of service levels of abstraction (Huhns and Singh 2005). This model is also similar to the one we described, although in our case the basic abstraction is provided by roles which can be provided by both components and services, whereas in this case the abstraction is provided on the basis of services only. In the latter, atomic services are used at the lowest level for forming subservices at the intermediate level, which are eventually used to compose high-level services.

Additional projects also aim to address the complete life cycle of QoS-aware applications. Similar to the MADAM approach, $2K^{Q+}$ (Duangdao et al. 2001) provides a QoS software engineering environment for specifying alternative component compositions and their QoS properties, which are then appropriately compiled for deployment on a specialized middleware. A platform-dependent compiler is provided, which produces executable code for reconfiguring the application at runtime by probing the QoS and resource availability.

The Quality Objects (QuO) framework (Loyall et al. 1998) relies on a suite of description languages for specifying QoS requirements. These specifications are compiled into executable code which is used for monitoring QoS and for controlling the interaction between distributed objects running on top of a CORBA-based middleware. This approach has the limitation that the specifications are platform-specific, as opposed to the composition and service plans we have described.

With concern to the SOC approach, many similarities are found in relation to our approach. For instance, SOC utilizes services as fundamental elements for developing applications and systems (Papazoglou and Georgakopoulos 2003). However, the area of adding context awareness to services is a new and promising one (Maamar et al. 2006). The proposed approach adds context awareness to some extent, although the entire decision making is centralized on the client side. However, more general approaches are currently under investigation.

Unlike the existing literature, the proposed approach offers a well-defined methodology for developing context-aware, adaptive applications. The underlying middleware supports automatic and autonomous reasoning on the possible adaptations for the selection of the most beneficial one. To the best of our knowledge, no related work proposes such a structured methodology for developing context-aware, adaptive applications. Additionally, a novel approach is proposed for the support of dynamically considering and exploiting services in the composition of component-based applications. Although in its infancy, this approach appears very promising and is one of the major points of focus of the MUSIC project (MUSIC Consortium).

6.7 Conclusions

This chapter described a novel methodology which utilizes an underlying middleware layer to ease the task of developing adaptive mobile applications. This methodology was partly studied and developed in the context of the MADAM project.

The proposed development methodology enables the design and implementation of adaptive applications for the mobile user, using the Separation-of-Concerns paradigm. In this approach, the developers are enabled to design the functional aspects of their applications independently of the specification of their extrafunctional behavior. An underlying middleware system is assumed to provide runtime support for the automatic self-adaptation of the deployed applications. It is argued that the proposed methodology, in combination with the provided middleware support, can significantly ease the effort required for the development of adaptive, mobile applications.

Additionally, it is important to state that although the MADAM results are novel and impact several software practitioners, more work is under way as part of a follow-up project: the Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments (MUSIC Consortium). Besides improving the existing results, MUSIC aims at providing additional support for ubiquitous computing environments, and also at providing better integration with SOA-based systems.

Acknowledgments

The work published in this chapter was partly funded by the European Community under the Sixth Framework Program, contracts FP6–4169 (IST-MADAM) and FP6–35166 (IST-MUSIC). Some of the ideas expressed in this chapter are the result of extensive collaboration among the partners of these projects. The authors would like to thank these partners and acknowledge the impact of their ideas on this work. The work reflects only the authors' views. The Community is not liable for any use that may be made of the information contained therein.

References

Alia, M., V. S. W. Eide, N. Paspallis, F. Eliassen, S. O. Hallsteinsen, and G. A. Papadopoulos. 2007a. A utility-based adaptivity model for mobile applications. In *Proceedings of the IEEE International Symposium on Ubisafe Computing* (UbiSafe), pp. 104–118. Niagara Falls, Canada.

Alia, M., S. Hallsteinsen, N. Paspallis, and F. Eliassen. 2007b. Managing distributed adaptation of mobile applications. In *Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems* (DAIS), pp. 556–563. Paphos, Cyprus.

Alpaydin, E. 2004. *Introduction to Machine Learning.* Cambridge, Mass.: MIT Press.

Amundsen, S., K. Lund, F. Eliassen, and R. Staehli. 2004. QuA: Platform-managed QoS for component architectures. Paper presented at the Norwegian Informatics Conference (NIK).

Arnold, K., J. Gosling, and D. Holmes. 2000. *The Java™ Programming Language*, 3rd ed. Upper Saddle River, N.J.: Prentice Hall.

Chen, G., and D. Kotz. 2000. A Survey of Context-Aware Mobile Computing Research. Technical Report TR2000–381. Department of Computer Science, Dartmouth College.

Costa, P., G. Coulson, C. Mascolo, G. P. Picco, and S. Zachariadis. 2005. The RUNES middleware: A reconfigurable component-based approach to networked embedded systems. In *Proceedings of the 16th International Symposium on Personal, Indoor and Mobile Radio Communications* (PIMRC), pp. 806–810. Berlin.

Dey, A. K. 2000. Providing Architectural Support for Building Context-Aware Applications. Ph.D. thesis, College of Computing, Georgia Institute of Technology.

Dey, A. K. 2001. Understanding and using context. *Personal and Ubiquitous Computing* 5, no. 1: 4–7.

Duangdao, W., K. Nehrstadt, X. Gu, and D. Xu. 2001. "2K^{Q+}: An integrated approach of QoS compilation and reconfigurable, component-based run-time middleware for the unified QoS management framework. In *Middlesex 2001: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, pp. 373–394. Heidelberg, Germany.

Floch, J., S. Hallsteinsen, F. Eliassen, E. Stav, K. Lund, and E. Gjørven. 2006. Using architecture models for runtime adaptability. *IEEE Software* 23, no. 2: 62–70.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston: Addison-Wesley.

Geihs, K., M. U. Khan, R. Reichle, A. Solberg, S. Hallsteinsen, and S. Merral. 2006. Modeling of component-based adaptive distributed applications. In *Dependable and Adaptive Distributed Systems (DADS Track): Proceedings of the 21st ACM Symposium on Applied Computing* (SAC), pp. 718–722. Dijon, France.

Horn, P. 2001. Autonomic Computing: IBM's Perspective on the State of Information Technology. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf. (Accessed November 30, 2007.)

Huhns, M. N., and M. P. Singh. 2005. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing* 9, no. 1: 75–81.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, LNCS 1241, pp. 220–242.

Lindholm, T., and F. Yellin. 1999. *The Java™ Virtual Machine Specification*, 2nd ed. Upper Saddle River, N.J.: Prentice Hall.

Lundesgaard, S. A., K. Lund, and F. Eliassen. 2006. Utilizing alternative application configurations in context- and QoS-aware mobile middleware. In *Proceedings of the 6th IFIP International Conference on Distributed Applications and Interoperable Systems* (DAIS). LNCS 4025, pp. 228–241. Bologna, Italy. Springer Verlag.

Loyall, J. P., D. E. Bakken, R. E. Schantz, J. A. Zinky, D. A. Karr, R. Vanegas, and K. R. Anderson. 1998. QoS aspect languages and their runtime integration. In *Proceedings of the 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers* (LCR), pp. 303–310. Pittsburgh, Penn.

Maamar, Z., D. Benslimane, and N. Narendra. 2006. What can context do for Web Services? *Communications of the ACM* 49, no. 12: 98–103.

MADAM Consortium. Mobility and Adaptation Enabling Middleware (MADAM). http://www.ist-madam.org.

Maes, P. 1987. Concepts and experiments in computational reflection. *ACM SIGPLAN Notices* 22, no. 12: 147–155.

McKinley, P. K., S. Masoud Sadjadi, E. P. Kasten, and B. H. C. Cheng. 2004a. Composing adaptive software. *IEEE Computer* 37, no. 7: 56–64.

McKinley, P. K., S. Masoud Sadjadi, E. P. Kasten, and B. H. C. Cheng. 2004b. A Taxonomy of Compositional Adaptation. Technical Report MSU-CSE-04–17. Department of Computer Science and Engineering, Michigan State University.

MUSIC Consortium. Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments (MUSIC). http://www.ist-music.eu.

Noble, B. 2000. System support for mobile, adaptive applications. *IEEE Personal Communications* 7, no. 1: 44–49.

Noble, B., and M. Satyanarayanan. 1999. Experiences with adaptive mobile applications in Odyssey. *Mobile Networks and Applications* 4, no. 4: 245–254.

Papazoglou, M. P., and D. Georgakopoulos. 2003. Service oriented computing: Introduction. *Communications of the ACM* 46, no. 10: 24–28.

Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, no. 12: 1053–1058.

Paspallis, N., A. Chimaris, and G. A. Papadopoulos. 2007. Experiences from developing a distributed context management system for enabling adaptivity. In *Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems* (DAIS), pp. 225–238. Paphos, Cyprus.

Paspallis, N., and G. A. Papadopoulos. 2006. An approach for developing adaptive, mobile applications with separation of concerns. In *Proceedings of the 30th Annual International Computer Software and Applications Conference* (COMPSAC), pp. 299–306. Chicago.

Picco, G. P., A. L. Murphy, and G.-C. Roman. 1999. LIME: Linda meets mobility. In *Proceedings of the 21st International Conference on Software Engineering* (ICSE), pp. 368–377. Los Angeles.

Satyanarayanan, M. 2001. Pervasive computing: Vision and challenges. *IEEE Personal Communications* 8, no. 4: 10–17.

Sousa, J. P., and D. Garlan. 2002. Aura: An architectural framework for user mobility in ubiquitous computing environments. In *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture*, pp. 29–43. Montreal.

Szyperski, C. 1997. *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Essex, England: Addison-Wesley.

Tennenhouse, D. L. 2000. Proactive computing. *Communications of the ACM* 43, no. 5: 43–50.

Walsh, W. E., G. Tesauro, J. O. Kephart, and R. Das. 2004. Utility functions in autonomic systems. In *Proceedings of the International Conference on Autonomic Computing* (ICAC), pp. 70–77. New York.

Weiser, M. 1993. Hot topics: Ubiquitous computing. *IEEE Computer* 26, no. 10: 71–72.

Zachariadis, S., and C. Mascolo. 2003. Adaptable mobile applications through SATIN: Exploiting logical mobility in mobile computing middleware. Paper presented at the 1st UK-UbiNet Workshop. London.