A Development Process for Context-Aware Adaptive Services

M. Autili, P. Di Benedetto, D. Di Ruscio, P. Inverardi, M. Tivoli
Dipartimento di Informatica
Università degli Studi di L'Aquila,
67100 L'Aquila, Italy
{marco.autili,paolo.dibenedetto,diruscio,inverard,tivoli}@di.univaq.it

Abstract

Pervasive computing infrastructure makes it possible for mobile users to run software services on extremely heterogeneous and resource-constrained mobile devices. Heterogeneity and device limitedness creates serious problems for the development and deployment of mobile services that are able to run properly on the execution context and are able to ensures that users experience the "best" Quality of Service possible according to their needs and specific contexts of use. In this paper we show how the main issues related to the development of self-adapting context-aware services are addressed in the IST PLASTIC Project with the support of CHAMELEON, a declarative framework for tailoring adaptable services.

1 Introduction

Nowadays, software services need to cope with variability, as services get deployed on an increasingly large diversity of computing platforms and operates in different execution environments. Heterogeneity of the underlying communication and computing infrastructure, mobility inducing changes to the execution environments and therefore changes to the availability of resources and continuously evolving requirements, require services to be adaptable according to the context changes.

Supporting the development and execution of such adaptable services raises numerous challenges that involve models, methods and tools. Integrated solutions to these challenges are the main targets of the IST PLASTIC project. The main goal of the PLASTIC project is the rapid and easy development/deployment of self-adapting services for B3G networks [**?**].

In this paper we introduce the PLASTIC development process model that proposes model-based solutions to address the main issues related to the development of self-adapting context-aware services. We instantiate (part of) the process model by providing a methodology to generate adaptable code from UML-based specifications. This methodology is supported by an integrated framework which is based on an UML profile of the PLASTIC domain. The UML profile has been defined in order to allow for the concrete modelling of the domain entities. This can be done by "mapping" the domain entities into stereotyped UML modeling constructs that can be then instantiated to concretely specify (in a conform way) PLASTIC service models.

This paper is structured as follows: Sec. 2 describes the proposed development process and Sec. 3 outlines the adopted framework supporting context-aware adaptation. The application of the approach is illustrated by a case study in Sec. 4. Finally, Sec. 5 briefly discusses related work and Sec. 6 gives some conclusions and future directions.

2 PLASTIC Development Process

In this section we present the PLASTIC development process model for self-adapting context-aware services. The adaptive nature of context-aware services makes unfeasible a standard approach to development and validation in which models are usually used to specify systems at different levels of abstraction before deployment activities. The ever growing complexity of software demands the specification of models which can assume crucial roles even at run-time to allow system adaptations and online validations.

With reference to Fig. 1, the square boxes represent software artifacts/models and ellipses represent activities. Lifecycle time goes from the top to the bottom of the figure. All the process activities originate from a `Conceptual Model` where entities and relationships of the context-aware services domain are defined [**?**, **?**]. Based on these entities, a `Service Model` can be specified in terms of its `Functional Specification` and its `Service Level Specification (SLS)`, i.e., its QoS characteristics.

Two main streams of activities originate from a `Service Model`, each addressing one of the issues

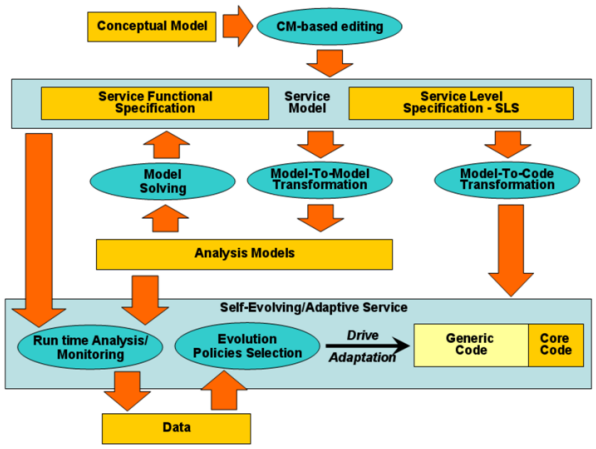

Figure 1. The PLASTIC process model

introduced above. On one side, `Model-to-Model Transformations` are devised in order to derive models for different kinds of analysis. Some of these models (e.g., stochastic behavioral models) are analyzed at development time to refine/validate the `Service Model` characteristics that the analysis addresses. This is represented in Fig. 1 from by the cycle: `Service Model` → `Model-To-Model Transformation` → `Analysis Models` → `Model Solving` → `Service Model`. On the other side, `Model-To-Code` transformations are used to build both the core code and the "generic code" of a service. The core code is the frozen unchanging portion of an adaptive service. The generic code is an adaptive code that embodies a certain degree of variability making it capable to evolve. This code portion is evolving in the sense that, basing on contextual information and possible changes of the user needs, the variability can be solved with a set of alternatives (i.e., different ways of implementing a service). A particular alternative might be suitable for a particular execution context and specified user needs. An alternative can be selected by exploiting the analysis models available at run-time and the service capabilities performing the `Run time Analysis/SLA Monitoring` and the `Evolution Policies Selection`.

Hereafter, we only focus on the right-hand side of the process model shown in Fig. 1. In this respect, the description of the adopted analysis and validation techniques is beyond the scope of this work. Thus, in Sec. 4, we illustrate the elements contained in the `Service Model` once analysis studies have been already performed, and show the adaptive service code which can be generated from a source `Service Model`. In the following section we describe the CHAMELEON framework that supports the context-aware adaptation phase of the PLASTIC development process. For the tools supporting all the other phases we refer to [**?**].

3 CHAMELEON

The framework CHAMELEON aims at developing and deploying (Java) adaptable service application. It supports the development of services that are generic and can be correctly adapted with respect to a dynamically provided context, which is characterized in terms of available (hardware or software) resources. To attack this problem we use a declarative and deductive approach that enables the construction of a generic adaptable service code and its correct adaptation with respect to a given execution context [**?**, **?**]. The CHAMELEON framework has been implemented [**?**] on the Java platform because of its widespread availability on today's mobile devices. Fig. 2 shows the components of the framework's architecture.

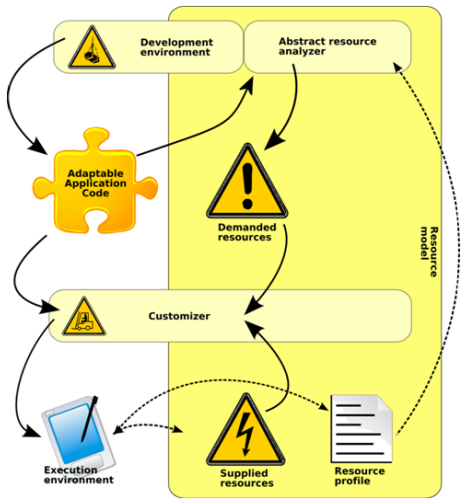

Figure 2. CHAMELEON Framework

The *Development Environment*, basing on the *CHAMELEON Programming Model*, is a standard Java development environment that provides developers with a means for easily specifying, in a flexible and declarative way, how the service can be adapted. Considering methods as the smallest building blocks that can be adapted in the service code, the programming model uses some ad-hoc extensions to the reference language, i.e., Java, to express adaptation (i.e. generic service code). Specifically, the standard Java syntax is enriched by dedicated key-word and annotations that permit to specify the following elements: *adaptable classes* that are classes declaring (without implementing) one or more *adaptable methods*; *adaptable methods* that are the entry-points for a behavior that can be adapted; finally *adaptation alternatives* that specify how one or more *adaptable methods* can actually be adapted. In general, it is possible to specify more than one alternative for a given adaptable class provided that for each adaptable method there exists at least one alternative that contains a

definition for it. In order to be conservative with respect to the existing tools, the development environment provides a preprocessor that takes as input generic service code and, by suitably combining the adaptable methods implementations specified by the various alternatives, produces a set of standard Java application alternatives that can be elaborated by traditional IDEs and compiled by traditional Java compilers.

The *Resource Model* is a formal model that allows the characterization of the computational resources needed to consume/provide a service. The Resource Model, enables the framework to reason on the set of adaptation alternatives and allows it to decide the "best-fit", depending on execution context information. A resource is modeled as a typed identifier that can be associated to natural, boolean or enumerated values. Natural values are used for consumable resources whose availability varies during execution (e.g., energy, heap space). Boolean values define non-consumable resources that can be present or not (e.g., function libraries, network radio interfaces) and enumerated values define non-consumable resources that provide a restricted set of admissible values (e.g., screen resolution, network type). The resource model permits to express both the resources needed to correctly execute an application alternative (*Resource Demand*) and the resources provided by a PLASTIC-enabled device[1] (*Resource Supply*). Resource demands and supplies are specified in terms of *resource sets* that couple resources to their values in the form $\{res_1(val_1), ..., res_n(val_n)\}$.

The *Abstract Resource Analyzer*, abstracting the behaviour of a standard JVM, statically inspects the Java bytecode of the different application alternatives and extracts a declarative description of their characteristics in terms of resource demands. The resource demands derives from both the annotations attached at source code and the resource consumption associated to bytecode instructions based on a *Resource Consumptions Profile*. The latter provides a characterization of the target execution environment, in terms of the impact that Java bytecode instructions have on the resources. Note that this impact depends on the execution environment since the same bytecode instruction may require different resources in different execution environments. It is out of the scope of this paper to go into details of our analysis technique, and we refer to [**?**, **?**] for further details.

The *Customizer* takes care of exploring the space of all the possible adaptation alternatives and carries out the actual adaptation before the deployment in the target environment for execution. The customizer, comparing the resource demands of the application alternatives with the resource supply of the execution environment, is able to propose the "best" suited application alternative and deliver (consumer- and/or provider-side) standard Java application that can be automatically deployed (via Over-The-Air (OTA) application provisioning) in the target devices for execution. The customizer bases on the notion of *compatibility* and *goodness*. Compatibility is used to decide if an application alternative can run safely on the requesting device, i.e., if for every resource demanded by the alternative a "sufficient amount" is supplied by the execution environment. Goodness expresses the effectiveness of each alternative with respect to resource consumption. It is based on a notion of priority among resources (e.g., energy might have a higher priority w.r.t. network bitrate) that is (by default) defined by the programmer and can be modified by users.

In [**?**] we describe a mechanism that, exploiting ad-hoc methods for *saving* and *restoring* the (current) application state, enables a form of applications' evolution (against monitored context changes and according to evolution policies) by dynamically un-deploying the no longer apt application alternative and subsequently (re-)deploying a new one with the desired aptitude.

The *Execution Environment* can be any device, equipped with a standard Java virtual machine, that will host the execution of the service. Typically the execution environment will be provided by Personal Digital Assistants (PDA), mobile phones, smart phones, etc. From this point of view, the Execution Environment is not strictly part of the framework we are presenting here. However it must provide the Resource Consumptions Profile and a declarative description of the resources it provides (i.e., the resource supply) that are retrieved by an additional software component deployed on the execution environment itself.

4 Case Study

In this section, the methodologies and tools previously discussed will be used for developing a PLASTIC e-Health Remote Diagnosis (RD) application whose requirement are specified in [**?**] and summarized in the following. The RD application is an eHealth (composite) service that allows to establish a link between patients and assistants providing support for video conferencing, medical agenda management, alarm generation and management, remote diagnosis, etc. Both professionals and patients are considered to be nomadic and can move with their mobile devices (e.g., move from outdoor to their office/home) in which case their multi-radio PLASTIC-enabled devices, through the PLASTIC middleware [**?**], handles the mobility transparently to ensure optimal connection (e.g., switches automatically from GPRS to WiFi connection since it provides better reception indoor). Therefore, mobility becomes a key issue and services need to be adapted, both to the heterogeneous networks capabilities and to the terminals that could

[1] A PLASTIC-enabled device is a device running specific components that, supported by the PLASTIC-middleware [**?**], are able to retrive context information.

be used by professionals and patients.

In the following, we only focus on the alarm management functionality (i.e., the `AlarmManagement` use case shown in Fig. 3) since it is complex enough to provide the reader with a full understanding of those PLASTIC development process activities concerning context-aware adaptation and their tool support outlined in Sec. 3. When an alarm is generated on the patient side due to some event (patient inactivity, dangerous vital parameters, help request) the e-Health system contacts one or more doctors to perform a diagnosis. On the doctor side the diagnosis process is supported by an RD consumer application that, connecting to an RD service provider installed on the patient side, allows the doctor to check the patient camera and monitor vital parameters (e.g., blood pressure, temperature, heart rate). The whole service is adapted according to both the patient (i.e., the provider) and doctor (i.e., the consumer) context.

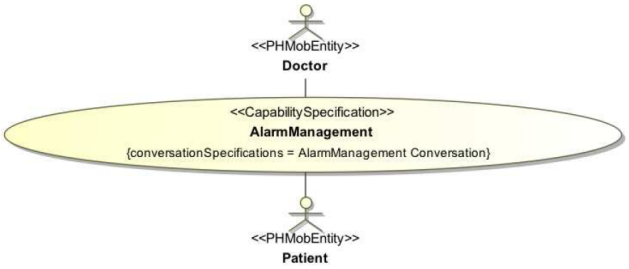

Figure 3. The `AlarmManagement` capability

By referring to Fig. 3, `AlarmManagement` is a `CapabilitySpecification` meaning that it represents a usage scenario of the RD application. A `CapabilitySpecification` is a UML2 use case linked to a `ConversationSpecification`. A `ConversationSpecification` models an orchestration of services that is specified to realize the behavior of the associated functionality. We will show the specification of the `AlarmManagement Conversation` later in this section. `Doctor` and `Patient` are the actors involved in the execution of the `AlarmManagement` capability. They are both stereotyped with `PHMobEntity` meaning that they can be mobile entities and, hence, a *mobility pattern* is associated to them (see later in this section).

Following the PLASTIC development process, the specification of the RD application proceeds with the definition of the services that have to be composed or orchestrated in order to perform the `AlarmManagement` capability. For this purpose, the *Service Description Diagram* in Fig. 4 is produced. By omitting details that are not crucial for the purposes the case study, a `ServiceDescription` stereotyped element models the interface of a service[2] as a set of `OperarionSpecification` stereotyped elements. An `OperarionSpecification` models an operation that can be invoked on the service[3].

`RDProviderService` models the interface of the service that runs on the patient's devices (e.g., desktop computer or PDA depending on the patient location). Note that it declares a `trigAlarm` operation that is invoked to send an alarm to the eHealth service provider where `eHealthService` runs (see also the `alarmManagement` operation of `eHealthService`). `RDProviderService` is a composite service (see the dependencies towards `DigitalCameraService` and `VitalParameterService` stereotyped with `Servicecomposition`) orchestrating all the management services for the electronic devices required to monitor the patient status. Depending on the patient mobility, these services will be either always available (e.g., at home) or partially available (e.g., if the patient moves out, the `DigitalCameraService` is not available). `eHealthService` is a composite service handling the alarm and the communication session among a patient and a doctor. It also handles the context switch of a patient or a doctor (e.g., when a doctor moves out the hospital or comes back to the hospital from outside) by keeping the patient-doctor session in a consistent state. `eHealthService` is composed by the `AssistantManager` service that is responsible to find a suitable doctor for the handling of a sent alarm. `RDConsumerService` is the service running on the doctor's devices. Among the other operations, its interface declares `connect`, `visualCheck`, and `vitalParameters` that are used when the doctor contacts the patient, wishes to get a picture or a video of what is happening at the patient side, and wants to monitor the patient vital status, respectively.

Once all services are defined, a number of *Business Process Descriptions* have to be provided. In particular, for each capability of a composite service (e.g., the `AlarmManagement` capability of the RD application), a business process model has to be specified in order to describe the interactions among the involved constituents (sub-)services. The specified business process models the conversation associated to that capability (e.g., the `AlarmManagement` conversation whose diagram is not shown for space reasons). Analogously with the web services domain, this diagram can be considered as a BPEL process model orchestrating the services shown in Fig. 4.

In general, a service can be implemented by one or more components and, in turn, a component can be used to implement one or more services. The PLASTIC approach proposes the use of the *Service Specification Diagram* for defining the components implementing basic services[4]. The

[2]It extends the semantics of a UML2 Interface.

[3]It extends the semantics of a UML2 Operation, e.g., to deal with all kinds of WSDL operations.

[4]They do not result in a composition/orchestration of other services.

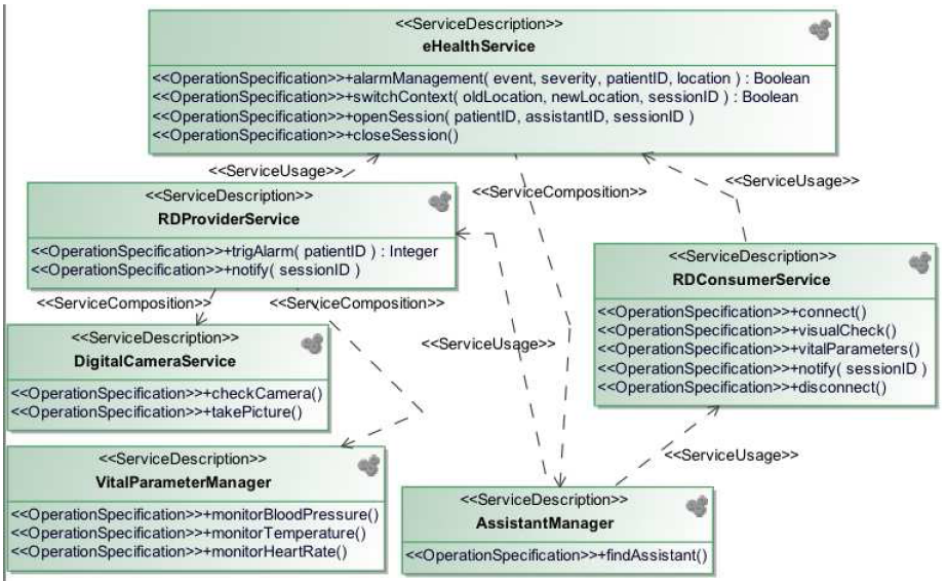

Figure 4. Service Description Diagram

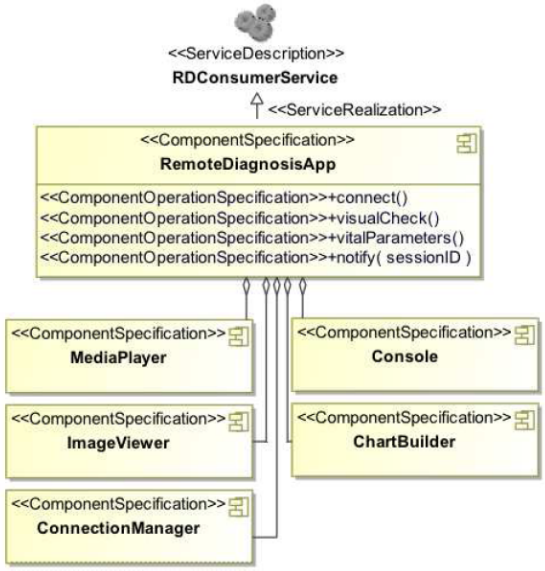

Figure 5. Service Specification Diagram for `RDConsumerService`

diagram in Fig. 5 specifies the components that implement `RDConsumerService` which will be deployed on the devices of the doctors. Hereafter we will focus on the implementation (and deployment) of `RDConsumerService` only since its operations (i.e., `connect`, `visualCheck`, and `vitalParameters`) require context-aware adaptations and, hence, for the purposes of the case study, `RDConsumerService` is one of the crucial services. According to Fig. 5, `RDCosumerService` is implemented by an aggregate component, i.e., `RemoteDiagnosisApp`. It uses other components (e.g., `ConnectionManager`, `MediaPlayer`, etc.) to implement the methods `connect`, `visualCheck`, and `vitalParameters`.

Once all the components which implement `RDConsumerService` have been identified, their interactions can be specified by means of the so-called *Elementary Service Dynamics Diagram*. This diagram is an extension of a UML2 sequence diagram and it models the messages that are exchanged, among the components implementing a service, when a service operation is invoked. The diagram, which is not shown for the sake of space, contains also non-functional annotations in order to enable performance analysis (e.g., worst-case execution time, latency, etc.).

In order to have a comprehensive description of `RDConsumerService`, each component implementing the service needs to be specified at a lower level of abstraction by means of a *Component Design Diagram* (see Fig. 6). In this diagram, the classes implementing the `RemoteDiagnosisApp` component are modelled. For the sake of clarity, in Fig. 6, we show only the main class among all the ones implementing `RemoteDiagnosisApp`. This class, called `RemoteDiagnosis` implements a J2ME MIDlet. The stereotype `AdaptableClass` is used to distinguish the classes whose methods' implementation is adaptable w.r.t. specified resources. For instance, `connect` is a method stereotyped with `AdaptableMethod` and it is adaptable w.r.t. the resources `NetBitrate` (i.e., network bitrate) and `Energy` (i.e., power consumption), as

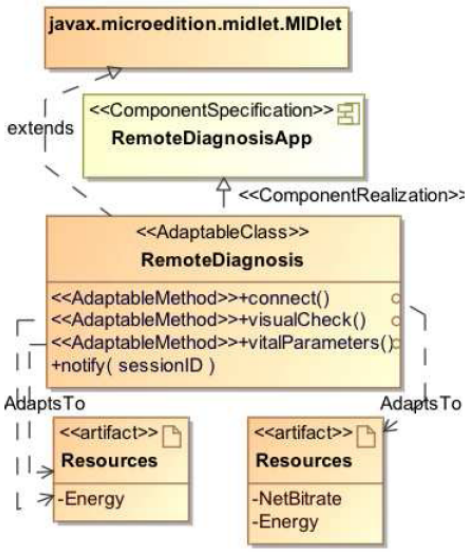

Figure 6. Component Design Diagram for `RemoteDiagnosisApp`

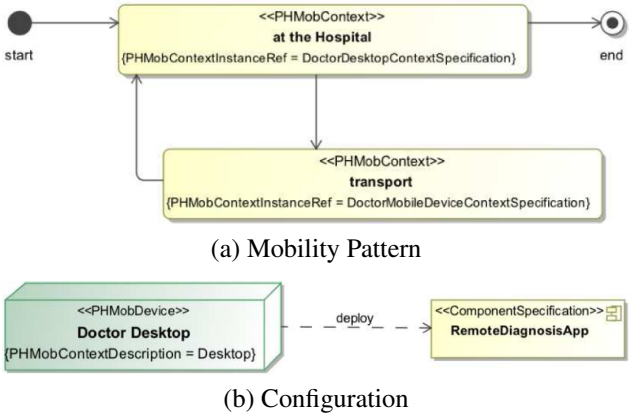

Figure 7. Physical Mobility Pattern Diagram for Doctor

specified by the dependency `AdaptsTo` from the `connect` method to the `Resources` artifact. `visualCheck` and `vitalParameters` have to be adapted w.r.t. only `Energy`. Component implementations can also be realized by non-adaptable classes (or non-adaptable methods). This is the case of components that do not require adaptation.

An `AdaptableMethod` can be adapted w.r.t. a set of resources that are a sub-set of the resources defining the different device contexts that can be reached by a mobile entity (e.g., a `PHMobEntity` such as `Doctor` shown in Fig. 3). The stereotype `PHMobEntity` internally declares a tagged value (`PHMpattern`) that points to the mobility pattern of the identified entity. Such mobility pattern is described by a *Physical Mobility Pattern Diagram* that is a UML2 state diagram where the states represent the system configurations (hardware plus software) the mobile entity has to deal with, while the arrows model the moving among configurations.

Focusing on the possible contexts where `RDConsumerService` can run, in Fig. 7.a we report the mobility pattern for the `Doctor`. At the hospital the doctor activates his device and the `RDConsumerService` starts its execution (`at the Hospital` state). During the working time, the doctor can move (e.g., from the surgery to the patient's house). Since the doctor service should be always available in the working time of the doctor, the service has to follow him also during his moves. This requirement implies an additional configuration that we model by means of the `transport` state.

The system configurations (or contexts) are described by deployment diagrams. One deployment diagram is associated to each different physical context. Nodes in the deployment are stereotyped with `PHMobContextDescription` having a tagged value (`ContextDescription`) pointing to the description of the device/network characteristics represented by the node. In Fig. 7.b we report the configuration of `RDConsumerService` when the doctor is at the hospital. For the case in which the doctor is moving on, an analogous deployment diagram has been specified where `RDConsumerService` is deployed on a mobile device. For the sake of brevity, this diagram is omitted.

The doctor runs `RDConsumerService` (actually the `RemoteDiagnosisApp` component implementing it) on his desktop computer when he is at the hospital, whereas `RDConsumerService` is run on a mobile device when the doctor moves. The characteristics of a particular device are specified, for resource-aware adaptation purposes, in terms of the resources that characterize the device. The characteristics of the doctor's desktop computer and mobile device are described by means of the diagram shown in Fig. 8. Resources which are not specified, as in the `Desktop` device, are assumed to be always available with the maximum value.

Resources are modeled according to the CHAMELEON resource model. At design-time, a device context is simply modeled as a set of resource identifiers that, at deployment time, will be associated to actual values to derive the device `resource supply`.

The last phase of the development process focuses on the code derivation of the resource-aware adaptable components implementing the services modeled for the RD application. Model-to-code transformations are used for this purpose. In particular, models are translated automatically into code skeletons by means of a developed code generator based on the *Eclipse Java Emitter Template framework* (part of the EMF framework [**?**]). JSP-like templates de-

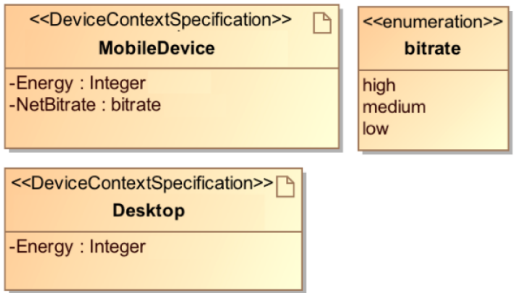

Figure 8. Device Contexts for `RDConsumerService`

fine explicitly the code structure and get the data they need from the UML model of the specified service exported into EMF. With this generation engine, the generated code can be customized and then re-generated without losing already defined customizations. Fig. 9 shows the generated skeleton code for the `RemoteDiagnosis` adaptable class shown in Fig. 6. Note that, adaptable methods are annotated with the resource w.r.t. which they adapt, to make easier developers' work.

```
1 adaptable public class RemoteDiagnosis extends MIDlet {
2   adaptable void connect();
3     /* adapts w.r.t. NetBitrate & Energy */
4   adaptable void visualCheck();
5     /* adapts w.r.t. Energy */
6   adaptable void vitalParameters();
7     /* adapts w.r.t. Energy */
8   ...
9 }
```

Figure 9. RemoteDiagnosis adaptable class skeleton

Alternative	Features	Resource Demand
Cons_1	Shows patient's camera images and textual data of vital parameters	{Energy(200)}
Cons_2	Shows video streaming and images from patient's cameras and draws diagrams of vital parameters	{NetBitrate(high), Energy(400)}

Table 1. Remote Diagnosis consumer application alternatives

Fig. 10 represents an excerpt of the generic code of the `RemoteDiagnosis` adaptable class written according to the CHAMELEON Programming Model (see Section 3). Note that, adaptable methods do not have a definition in the adaptable class (see the keyword `adaptable`) where they are declared but they are implemented within *adaptation alternatives* defined through the keywords `alternative` and `adapts`.

```
1 adaptable public class RemoteDiagnosis extends MIDlet {
2   ...
3   adaptable void connect();
4   adaptable void visualCheck();
5   adaptable void vitalParameters();
6
7   ... /* Others non adaptable methods */
8 }
9
10 alternative class Powerful adapts RemoteDiagnosis {
11   private void connect() { ...
12     Annotation.resourceAnnotation(``NetBitrate(high)'');
13     QoSInfo.setBitrate(HIGH);
14     PlasticMiddleware.selectNetwork(QosInfo);
15   }
16   private void visualCheck() {
17     /*shows video streaming from patient's cameras*/}
18   private void vitalParameters() {
19     /*draws diagrams of vital parameters*/}
20 }
21
22 alternative class Limited adapts RemoteDiagnosis {
23   private void connect() {  ...
24     QoSInfo.setBitrate(LOW);
25     PlasticMiddleware.selectNetwork(QosInfo);
26   }
27   private void visualCheck() {
28     /* shows patient's camera images */ }
29   private void vitalParameters() {
30     /*shows textual data of vital parameters*/}
31 }
```

Figure 10. RemoteDiagnosis adaptable class

The Programming Model permits to specify *Annotations* that can be used to attach resource information to methods (see the keyword `Annotation`). Annotations can be specified by calls to "do nothing" static methods of the `Annotation` class. For instance, in Fig. 10 the method call `Annotation.resourceAnnotation (``NetBitrate(high)'')` is used to specify that the `Powerful` alternative demands for a high network bitrate. That annotation will contribute to determine the overall application resource demand.

The generic code in Fig. 10 will be preprocessed by the *CHAMELEON Preprocessor* and the two standard Java application alternatives described in Table 1 will be derived. Such table reports also the resource demand associated to those application alternatives calculated by the *CHAMELEON Analyzer*. For example, the resource demand of the Cons_2 application alternative specifies that, to run safely, the alternative will require a high network bitrate (`NetBitrate(high)`) and a battery state-of-charge of the target device at least of 400 energy unit (`Energy(400)`). At deployment time those resource demands will be compared by the CHAMELEON Customizer with the resources supplied by the doctor device and the most suitable application alternative will be deployed on the doctor device. Further details on the derivation steps and on how all the derived artifacts are used at deployment- to runtime (to tackle adaptation) can be found in [**?**].

5 Related Work

For sake of space, we cannot address all the recent related works in the wide domain of PLASTIC project, thus in the following we provide only some major references. Current (Web-)service development technologies, e.g., [**?**] (just to cite some) do not take into account context-awareness. Our process borrows concepts from these well assessed technologies and builds on them to take into account context-awareness of services for self-adaptiveness purposes. We can also relate to other approaches to resource-oriented analysis, such as [**?**, **?**].All these approaches give a resource model as we do, but differently from us, they do not use these models in the context of adaptation and, moreover, they do not provide the flexibility for reasoning on the resources in the perspective of program adaptation. In [**?**], supported by the MUSIC project[5], the authors propose the design of a middleware- and architectural-based approach to support the dynamic adaptation and reconfiguration of the components and service composition structure. Similarly to us, the idea for the designed approach is based on requested and offered QoS, and supports SLA negotiation. The work in [**?**] presents CARISMA, a mobile middleware that facilitates the development of adaptable and context-aware mobile applications. CARISMA handles context changes using policies and achieves dynamic adaptation through reflection. The middleware is abstracted to the applications as a service provider that can be customized through metadata encoding middleware behaviour. Differently from us, the approaches in [**?**, **?**] do not tackle adaptability to the device execution context through resource oriented analysis, parametric w.r.t. its resource consumption profile.

6 Conclusions and Future Work

This paper presents the development process model defined in the context of the IST EU PLASTIC project [**?**] which aims at offering a comprehensive provisioning platform for context-aware and adaptable software services. In particular, this work describes the instantiation of the process model within an UML world. Models and techniques for developing in UML adaptive code of context-aware services which have to show optimal QoS within different contexts have been integrated. The approach is supported by the CHAMELEON framework conceived to automate those process steps concerning resource-aware adaptation.

As future work, we plan to investigate the usage of non-UML methodologies and tools within the process model, such as formal (functional and non-functional) specification of services. This would allow to introduce in the process formal refinement and analysis techniques, such as model checking. The application of the approach to a real world case study would further allow to refine and validate the whole framework.

[5] http://www.ist-music.eu/

Acknowledgments.

This work is part of the IST PLASTIC project funded by the European Commission, FP6 contract number 026955, http://www.ist-plastic.org/.

References

[1] A-MUSE Project. Methodological Framework for Freeband Services Development, 2004.

[2] E. Albert, S. Genaim, and M. Gomez-Zamalloa. Heap Space Analysis of Java Bytecode. In *ISMM'07*, Oct. 2007.

[3] M. Autili, P. D. Benedetto, P. Inverardi, and F. Mancinelli. CHAMELEON project - SEA group. http://www.di.univaq.it/chameleon/.

[4] M. Autili, P. D. Benedetto, P. Inverardi, and D. A. Tamburri. Towards Self-evolving Context-aware Services. In *Proc. of CAMPUS'08 (DisCoTec'08)*, volume 11, 2008.

[5] M. Autili, V. Cortellessa, A. D. Marco, and P. Inverardi. A Conceptual Model for Adaptable Context-aware Services. In *WS-MATE*, 2006.

[6] G. Barthe. MOBIUS, Securing the Next Generation of Java-Based Global Computers. ERCIM News, 2005.

[7] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework*. Addison Wesley, 2003.

[8] L. Capra, W. Emmerich, and C. Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE TSE*, 29(10):929–945, Sep 2003.

[9] P. Inverardi, F. Mancinelli, and M. Nesi. A Declarative Framework for adaptable applications in heterogeneous environments. In *ACM SAC*, 2004.

[10] M. Autili, P. Di Benedetto, P. Inverardi, F. Mancinelli. A resource-oriented static analysis approach to adaptable java applications. In *Proc. of CORCS'08 (COMPSAC'08)*, 2008.

[11] F. Mancinelli and P. Inverardi. Quantitative resource-oriented analysis of java (adaptable) applications. In *WOSP'07*, pages 15–25, NY, USA, 2007.

[12] PLASTIC Project. D1.2: Formal description of the PLASTIC conceptual model and of its relationship with the PLASTIC platform toolset.

[13] PLASTIC Project. DoW, 2005. http://www.ist-plastic.org.

[14] PLASTIC Project. D3.3 - Middleware: Assessment and Revision, 2007.

[15] PLASTIC Project. D5.1: Integrated PLASTIC platform and supporting user guide and training courses, 2007.

[16] R. Rouvoy, F. Eliassen, J. Floch, S. O. Hallsteinsen, and E. Stav. Composing components and services using a planning-based adaptation middleware. In *SC*, LNCS, pages 52–67, 2008.