Adaptation Design in Ubiquitous Computing

Michael Fahrmair, Christian Leuxner, Wassiou Sitou, Bernd Spanfelner

TUM-I0801
Januar 08

TUM-INFO-01-I0801-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Abstract

The concept of *Ubiquitous Computing*, also called *Ubicomp* for brevity, describes a new paradigm on the usage of computer based systems, which is – in comparison to the usage of conventional computers – characterized by an enhanced degree of user centering, thus enabling users to benefit from computer usage and support in as many situations as possible. The computer as an apparent tool thereby steps into the background while the actual needs and wishes of the current user step into the foreground. As usability plays the major role in Ubicomp, accomplishing user needs must be done automatically, i.e. without capturing a user's attention.

Despite many years of research, designing successful applications for Ubicomp is still a complex and error prone task. This is because most work only concentrates on technical implementation of context awareness as an enabling technology for realizing ubiquitous systems. But apparently, adapting to satisfy user needs in varying environments will also require an extension of traditional design methods and processes. We therefore propose a methodological approach to adaptation design. The proposed methodology explicitly handles the notion of context dependent system flexibility that is needed in many models and methods used for adaptive system design. This iterative approach enables system designers to invent systems that are context aware and change their behavior appropriately to a user's intentions – independent from designing individual components or services.

As a proof of concept this methodical approach was evaluated by means of two long-term case studies: within one case study an industrial hospital setting exposing certain self healing skills was developed while the other setting concerns a Context Aware Task Scheduler (CATS), which aims at assisting business users by optimally arranging their daily schedules. For that purpose, the CATS inter alia makes context depending decisions and adapts relevant system functionalities according to the current usage situation, e.g. by automatically shifting to silent notification when the user is currently attending a conference.

Contents

1 Introduction

One of the main aspects that differentiates ubiquitous systems from conventional computer based systems is their ability to automatically identify user needs. This implies that for certain system functions no direct or conscious interactions by means of menu dialogs or command interpreters are necessary for communicating the actual situation and associated needs, respectively. This ability includes particularly the identification of *situational* user needs [1, 2], which are merely valid in specific user situations such as driving a car. *Context adaptation* in this context can be understood as an explicit handling of situational user needs and hence enables system usage in situations where most other existing computer systems are useless, e.g. in situations where the user can not or does not want to use an explicit command interface. To do so, context adaptive systems need to be aware of their *context* and use it to differentiate between certain situations. A context adaptive messaging service for instance should recognize that the user is currently driving a car (context) and in consequence read an important incoming email to the user, instead of displaying it on the on-board panel (adaptation). When on the other hand the driver is currently navigating within a highly frequented metropolis like New York for the first time, trying desperately to follow the suggested directions, the system better delays an incoming message until later for not distracting the user from his current tasks.

Fundamental research in this area has been done in constructing frameworks and prototypes concerning adaptive systems [3, 4, 5]. The important process of *systematically* designing context as well as context adaptive system behavior itself however is typically neglected [6]. Although first works on dealing with adaptation and context-awareness by means of process support, such as the method for personal and contextual requirements (PC-RE) by Sutcliffle et al. [2] or the scenario based approach proposed by Kolos-Mazuryk et al. [7], have been introduced, there is still a lack of *integrated* processes, supporting the development of context adaptive applications in Ubiquitous Computing. In addition a merely intuitive yet unsystematic modeling of context adaptation often causes ubiquitous applications to fall into the trap of *unwanted behavior* (UB)[8] (i.e. assumptions in the environment or user model become wrong, which leads to unexpected or unwanted system behavior). Therefore the legendary vision of Mark Weiser [9] still fails to become reality.

In our point of view this shortcoming should be addressed by systematic engineering approaches. The methodology proposed in this paper results from over seven years research experience in the field ubiquitous and mobile computing that has its main focus on the engineering of context and adaptation behavior.

In this paper we give a brief overview of the core steps of the suggested methodology (section 3). These steps ease the process of complex context construction, the elaboration of arbitrary user needs, the specification of adaptation decision logics, and their realization and deployment within an adaptation framework. For better understanding of the basics, a recent version of this framework (building on top of a service oriented reconfigurable component architecture) is briefly introduced together with some common sense definitions in section 2. Furthermore in section 4 we present the first results of a long term case study that is about an autonomous task scheduler (the **C**ontext **A**ware **T**ask **S**cheduler) negotiating and arranging the user's tasks appropriately. This example is not really considered brand new (e.g. [10] or [11]), some might even argue that it is quite boring due to the great number of existing implementations concerning the same purpose. However it makes up a perfect example because there are so many case studies to compare against and, honestly, we never saw one that really worked in real life. One could suspect that the reason for this shortcoming is that this challenge cannot be solved at all and trying to make such a system work is tedious and futile after all. Yet this case study mainly served the purpose of evaluating and extending the methodology; hence choosing the worst case scenario seemed a good idea to us. However the proposed methodology of course is also appropriate for designing and implementing arbitrary (and easier to accomplish) ubiquitous systems. For further evaluations, we currently apply the proposed methodology in two industrial case studies, both in a hospital setting.

2 Foundations

After motivating the need for an extended methodology for designing ubiquitous systems and before presenting a detailed explanation of that design methodology in section 3, some important concepts on the topic are introduced in the following. Unfortunately we experienced that related terms concerning concepts like ubiquitous, pervasive, ambient or mobile computing – to name just a few – considered in previous literature often suffer from being used for very different concepts. We do not attempt to abandon the heap of definitions currently in usage – a concise summary of related terms after all can be found in [4]; but we introduce some generic definitions which are mandatory for the understanding of this paper in order to provide a common understanding of considered terms.

2.1 Related terms

As stated previously the notion of Ubiquitous Computing is very central to this paper, as we aim at providing a successful approach towards the design of ubiquitous applications. We adopt a definition for Ubicomp, which is generic enough for covering the essence of previous definitions we found so far.

Definition

☞ Ubiquitous Computing denotes the direct or indirect usage of computer based applications in as many situations (e.g. locations, time, activities) of a user as possible.

We use the term *indirect usage* to denote that one or more system inputs comprise contextual information gathered from the system environment by means of sensors, as opposed to user inputs which are entered *directly*. The above definition focuses on the *usability* of functions. A functionality is usable, if the necessary hard- and software is available (availability), it furthermore satisfies the current user needs (applicability) and the user can operate the functionality according to his current activity (operability). The thesis of Schmidt [5] clearly indicates that Ubiquitous Computing always means computing in a certain *context*, which directly leads to the definition of an equally central concept.

Definition

☞ Context is the sufficiently exact characterization of a situation by means of information that is both perceivable by the system and relevant for the adaptation of the system.

Perhaps we should emphasize that our notion of context, unlike most typical definitions which merely comprise *external* information perceivable from the system environment, needs to be understood in a broader sense: we use context as a generic container for *arbitrary* information concerning the user, her operational environment or the inner state of an ubiquitous application. By means of context we furthermore achieve a decoupling of communicating components as illustrated in figure 2.1. This decoupling mechanisms enables a system design which facilitates communication between a priori unknown and unreliable components, as they occur in mobile and highly dynamic networks. Context elements thereby handle the communication by buffering the exchanged information, thus enabling their discovery at runtime as well as any desired manipulation of the buffered information between it's generation and usage.

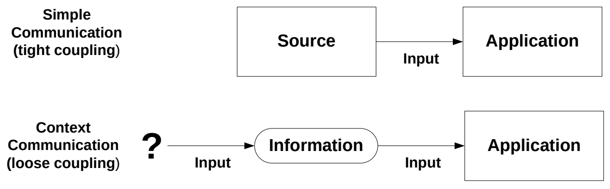

Simple Communication (tight coupling)
Source
Input
Application
Context Communication (loose coupling)
?
Input
Information
Input
Application

Fig. 2.1: Decoupling of components by means of context

The elicitation of context (characterizing the current usage situation) by means of sensors is indeed the essential precondition for building ubiquitous systems. Because without adequate sensors which observe the behavior and activities of a user and her corresponding environment, the whole human-machine-communication must be accomplished via direct user input/output by means of traditional user interfaces (e.g. menus, command interpreters etc.). However, an exclusive and direct user interaction is impractical for many usage situations in everyday life (e.g. when driving a car). Hence a promising approach for enhancing the system usage to situations, in which most conventional applications remain idle due to their limited interaction possibilities, is to *adapt* the system behavior (e.g. change the current input/output mode or applying inputs on behalf of the user that are derived or deduced from sensor inputs) in order to reflect the current usage situation and associated (situational) user needs. The following definition embraces this approach. For the purpose of this paper, the definition of context adaptation is sufficiently exact. However, it does not provide a criterion for definitely delimiting systems which are context adaptive from those who are not. We are currently working on a formal definition of context adaptive systems to overcome this shortcoming.

Definition

☞ Context adaptation is an automated adjustment of the observable behavior or the inner state of the system according to a context.

Automated in this context means, that the adaptation itself is accomplished without user interaction. To emphasize this: it seems obvious that context adaptivity contributes very little without being executed automated. The reason for this observation is, that beside being able to recognize and evaluate different usage situations, ubiquitous systems in addition need to make the appropriate adaptation decisions *on their own* in order to satisfy the usability requirements mentioned above, i.e. to enable a system usage in as many situations as possible. If for example a driver is currently unable to read an incoming message on the on board display due to a critical traffic situation, he will neither be able to navigate through a user menu in order to find the function responsible for activating the speech output; the system is supposed to activate this function by itself – automated. The adaptation process thereby typically involves the following three steps, which are illustrated in figure 2.2 as well.

1. *Sensors* collect relevant information (concerning a current situation) from the operational environment and store these in decoupled *context*.
2. *Interpreters* then recognize a situation based on the current context state and, if necessary, calculate an appropriate adaptation decision regarding this situation.
3. *Actuators* execute the calculated adaptation which mostly results in a changed (reconfigured) system behavior to satisfy current user needs.

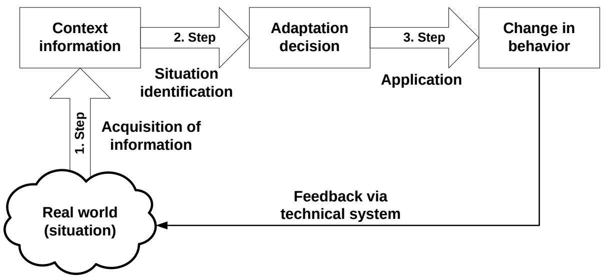


Fig. 2.2: Sub-processes within a context adaptation

In case the adapted system behavior differs from the expectations of the user, i.e. the context adaptation failed to satisfy the actual user needs and exposed *unwanted behavior* as described in [8], a compensation mechanism called *calibration* is proposed in [4], which allows users to effectively intervene the adaptation process by adapting the context adaptation itself. The particular services involved in the adaptation process described above are all elements of a well founded formal specification technique named *calibrateable context adaptation model*, shortly *K-Model* or *adaptation model*, which is introduced in the following section. A detailed description of the calibration mechanism can be found in section 2.3.

2.2 Calibrateable context adaptation model

Several architectures and fewer modeling techniques have been evolved for designing context adaptive systems, whereby a couple of selected approaches are shortly summarized in [3] together with their associated pros and cons. The general idea of our modeling approach is to provide system designers with a clear and structured notation for explicitly representing the context adaptive system behavior, which can be communicated to end users. The principle of this notation is to make both the modular structure and the workflow of the adaptation subsystem explicit in such a way that reasoning about the modeled system behavior is facilitated. This approach so to say brings up the decision logic, which in case of conventional systems remains unalterable inside a black-box component, thus effectively written in stone. In contrast we recommend to express the adaptive system behavior by means of adaptation models or shortly K-Models, which are formally founded on the component-oriented FOCUS theory introduced in [12].

The development of the proposed modeling technique and it's corresponding implementation on the basis of a framework (section 2.4) is motivated by an observation that was made over and over deploying ubiquitous systems within real world environments: despite of running perfectly under laboratory conditions, these prototypes usually exposed some kind of unexpected behavior when deployed in the wild, even though the underlying specification was accurately implemented. Hence this phenomenon can not be reduced to classical implementation defects. Since no established notion concerning this observation exists, we simply call it *Unwanted Behavior* or shortly *UB* [8]. The reasons for Unwanted Behavior can on the one side originate from insufficient Requirements Engineering (RE), in which certain user needs and usage situations are overlooked. On the other side does even the most sophisticated RE process ultimately result in a requirements specification, which is an abstraction based on static assumption made at some stage in the development process. However this inherently *static* abstraction is consequently subject to the Frame Problem [13] known from AI. In either case does the system model generated at design time not comply with the mental model, the user is currently associating with the considered system. In consequence a system behavior is exposed, which differs at least for certain usage situations from what the user would expect. Since the Frame Problem is still an open – and eventually unsolvable – issue, we propose the K-Model and it's runtime calibration as an efficient mechanism for circumventing this issue.

The K-Model uses only four basic elements for describing an arbitrary complex and adaptive system behavior. These elements structure all possible services of the adaptation subsystem into the four service types *sensors*, *interpreters*, *actuators* and *context elements*, which exhibit a type-specific behavior on their own. Currently two representation forms for documenting K-Models exist:

- For the purpose of designing and communicating adaptation models, a graphical notation augmented by different annotations and abstraction techniques was invented, which enhance the readability of adaptation models.

- A machine-readable representation in form of a platform independent XML file was chosen for implementing adaptation models via the CAWAR framework (section 2.4).

The four basic service types of an adaptation model are described in the following sections.

Sensors

Sensors are responsible for retrieving relevant information from inside and outside the system. They accomplish this task by writing sensed information to dedicated context elements, which in turn act as information buffers. Sensors within an adaptation model qualify to model physical sensors like thermometers, movement and light sensors as well as internal or external software entities that enter information into the system's context (e.g. a remote web service) or even human beings, since terminal inputs may also be treated as perceivable context information.

Context elements

Context elements act as buffers for storing arbitrary information. In addition they decouple the three further service types (sensor, interpreter and actuator), since a direct communication without context is not allowed for any of these service types. Depending on the adjacent service types and their associated semantics, a context element may represent measured context data (written by a *sensor*), combined and interpreted information (arranged by an *interpreter*) as well as resulting adaptation decisions which are gathered and implemented by an *actuator* (see figure 2.3).

Interpreters

Interpreters are the information processing entities within an adaptation model. They both gather input in the form of context elements and store the processed information to context elements. The way how interpreters transform their gathered input relies on the underlying logic the certain interpreter exposes and may embrace a simple data forwarding as well as an arbitrary complex interpretation logic (e.g. rule engine, neuronal network). Interpreters aquire the decision making within the adaptation subsystem and hence may expose certain learning capabilities.

Actuators

Finally actuators are responsible for implementing a calculated context adaptation by triggering the control components within the core system or the environment, which in turn change the system behavior according to the resulting adaptation decision. The context representing this adaptation decision is usually derived from perceivable sensor data, which is appropriately composed by interpreters as described above. For better differentiation the context elements coupled with actuators are also called *adaptation context.*

The overall process of context adaptation by means of a K-Model containing the just described service types is illustrated in figure 2.3. The typical activities involved in the adaptation process thereby correspond to these already depicted in figure 2.2, namely the *acquisition of context* (step 1), the *situation identification* (step 2) and the *application of the adaptation decision* (step 3).

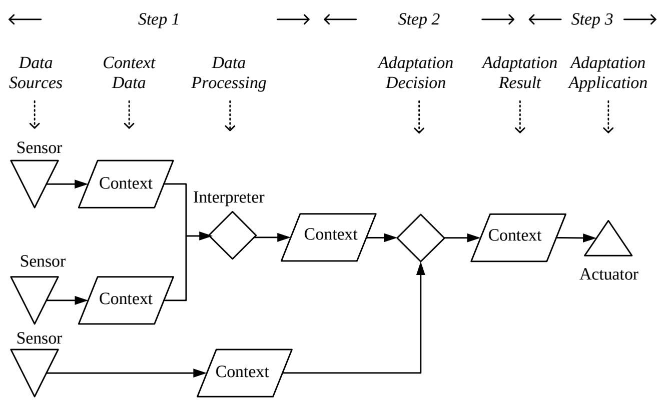


Fig. 2.3: Context adaptation by means of an adaptation model

As previously stated, these four elements suffice to represent an arbitrary adaptive system behavior. In order to achieve a better structuring of the model, which is easier to read and maintain, it is often beneficial to annotate certain elements like context and interpreters. The context space is thereby separated into a) *sensor context* denoting any information that directly originates from a sensor, b) *situation context* containing all information necessary in order to identify the current usage situation on basis of sensor context and c) *adaptation context* which ultimately comprise the decision about the required adaptation.

Designated interpreters are analogously annotated as *situation adaptors*, which typically combine context information (sensor and situation context) in order to identify a sufficiently exact abstraction of the current usage situation. *Adaptation actions* are another annotation possibility, which helps to model the *situational* requirements occurring in a certain usage situation – represented by a situation adaptor. As opposed to a situation adaptor, an adaptation action is a rather abstract modeling concept (i.e. placeholder) which needs to be refined by one or more basic elements (e.g. a single context element or an interpretation chain), as soon as the exact model representation for this situational requirement is at hand.

It should already be mentioned, that all elements contained in a K-Model are solely *services*; in order to use a contained service, it previously needs to be bound to a actual component fulfilling this service. For maximum software flexibility this service-component coupling may be delayed until runtime, after appropriate components fulfilling the specified services are discovered. This proceeding is of particular importance when employing external resources only available in certain usage situations: consider an external monitor (environment component), which is detected by the ubiquitous system hosted on a Pocket PC when entering the room. This monitor is bound to a specified display service for the duration the Pocket PC resides within this room. By the time the user – who is carrying the Pocket PC – is leaving the room, the monitor gets out of range; hence the display service falls back upon it's default display of the Pocket PC. By exchanging the currently available components (*reconfiguration*), the system adapts itself to provide the most appropriate resources currently available, thus enabling a usage in as many situations as possible.

The following definition outlines the main characteristics of a K-Model. An *activator* represents a special actuator, which inter alia identifies and binds components to specified services.

Definition

☞ A K-Model is a model of a calibrateable context adaption containing an activator that acts on a set of sensors, interpreters, actuators and context elements [4].

2.3 Calibration

Ubiquitous systems are prone to a phenomenon called *Unwanted Behavior* (UB) [8]. This phenomenon denotes situations, in which the observed system behavior differs from what the user would expect. The reason for this phenomenon is a divergence between the system model (which represents the mental model of the system designer) and the current mental model of the user. Such divergences may be caused by an insufficient Requirements Engineering or a more substantial problem called *frame problem* [13]. In any case does the ubiquitous application, when exposing UB, not serve the current user needs and in a best case is useless; at worst it may react obstructive or even harmful. To circumvent this problem, which was over and over observed when deploying ubiquitous systems in the wild, a concept called *calibration* is introduced. It enables an explicit adjustment of the systems adaptation behavior in case UB occurs; it does however *not* try to solve the frame problem in general. But since the frame problem seems to have no solution, calibration is an effective way to deal with UB.

The fundamental idea of the calibration concept is to create an explicit model of the adaptive system behavior, e.g. by means of the K-Model. This explicit modeling indeed is the basis for making the adaptive system behavior itself accessible to another adaptation, namely the calibration. As deficiencies in the model can not be recognized from within the model, a higher instance is needed for reasoning about the system model, i.e. for deciding if the model accomplishes what it is supposed to do. This higher instance can in turn be another system as well. Consequently the system representing this higher instance will be prone to UB as well. In order to break this chain, the last instance for reasoning about a considered system model always has to be a *user*.

Calibration can be used to correct system behavior both after an UB already occurred and in cases an upcoming UB can be predicted due to knowledge about the adaptive behavior and upcoming situations. To achieve a total system reconfiguration by means of calibration, the system model must also comprise a self-description of all contained services, as in the case of K-Models. We refer to [4] for a detailed description of the calibration mechanism.

2.4 The CAWAR framework in a nutshell

The CAWAR framework is a generic approach to support all kinds of adaptation in reconfigurable systems. Selected aspects of this framework, which are necessary for the understanding of how context adaptation models are technically realized, are outlined in the following sections. For a detailed conceptually introduction of the framework we refer to [14], whereas a description of it's technical realization can be found in [15].

2.4.1 Framework overview

In this section a short overview about the overall CAWAR (Context AWare ARchitectures) framework is given, while the two subsequent sections discuss certain framework concepts in more detail. The framework principally consists of the following elements:

1. A set of *components* comprising the technical implementation of typical infrastructure functionality, i.e. context storage, discovery, etc.
2. A set of low level *interfaces* (API), that provide the most generic abstraction of context management, i.e. sensors produce context, actuators consume context, etc.
3. A *reference architecture* that suggests a basic generic pattern of how a context adaptive system can be designed in a completely reconfigurable way – formally context adaptation can be understand as a self reconfiguring filter [4]. Following that pattern, any implementation of a context adaptive application can serve as a framework for bootstrapping any other context adaptive application.

Components, interfaces and architecture together form a basic framework for context awareness and adaptive applications. To develop a certain application, the framework merely must be fed with the desired system behavior in form of an adaptation model, whereby the principles for designing adaptation models are described in section 3. Furthermore the components fulfilling the respective application services must be made available to the framework. However provided a proper discovery mechanism such components can be detected and bound at runtime.

The framework initialization is conducted by a designated actuator component named *model activator*, which expects a list of all required services (logical service descriptions) and a list of (currently) available components (technical realizations or references) from the context itself. Such a description is e.g. given by a XML file representing the adaptation behavior of the considered system, i.e. the K-Model. This model has to be previously read by a special sensor and written into an appropriate context element. The context comprising the K-Model can be further processed – allowing for self introspection and self adaptation – before it is ultimately deployed by the model activator. The latter finally reorganizes the services (sensors, interpreters, etc.) as well as their corresponding component bindings (if available), thus reconfiguring the system in order to technically implement the K-Model.

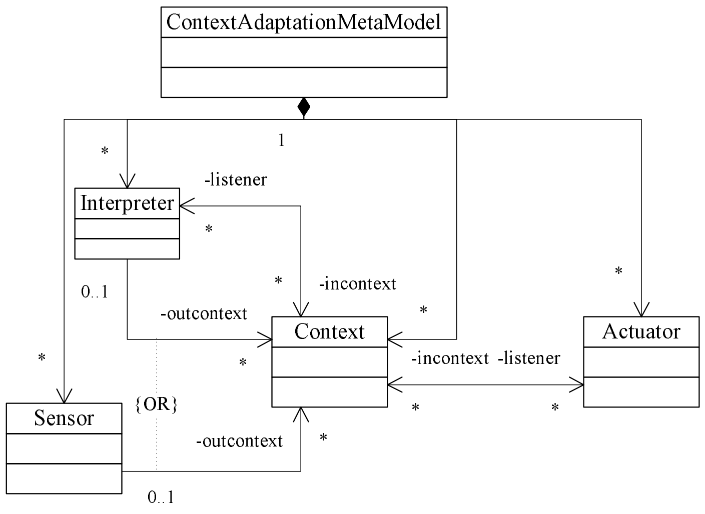


Fig. 2.4: Meta model describing the principle structure of the K-Model

The underlying meta model of any K-Model, which describes the principle composition of it's logical service architecture – so to speak it's grammar – is illustrated in figure 2.4. A concrete K-Model, containing the particular services which constitute the adaptive behavior of a considered application, is indeed an *instance* of the depicted meta model, which can be read by the model activator for initializing the context adaptive system as mentioned above.

2.4.2 Syntactical and semantical types

As previously mentioned are all services contained in a K-Model specified by a logical service description. Each description thereby contains a syntactical (syType) and semantical (smType) description of the service they provide. Both types are used to match a suitable technical component that could implement the specified service. In other words do syType and smType specify which components the service could possibly route messages to.

SyType describes any higher level protocol the service implementing component should understand using the standardized low level interfaces of sensors, interpreters, actuators and context elements, including at least the data format accepted. Moreover it could contain any other technical information needed to reduce the number of matching components e.g. QoS parameters, billing information etc. SyType should contain information needed by the activator to contact and bind possible candidates or it contains even the reference to a single component instance ensuring that only one specific component will match the description.

SmType in contrast describes the meaning or usage intention of a certain component instance besides its technical characteristics. Usually this can be used to distinguish between several instances of technical identical components. For example there could be several identical temperature sensors or terminals connected to a single system. However they can have different meanings regarding to the context like outside temperature, inside temperature, kitchen terminal or entrance terminal. A syntactical description is insufficient in this case since it could match more than one component instance. In order for the activator to distinguish which component instance should be bound to e.g. a sensor that delivers an "outside temperature" a smType can be used. One of the available sensors needs to have been marked with a meaning of "outside temperature" as well. Note that semantical marking is specific to the application scenario and hence part of the context and one of the main tasks of calibration.

It should be mentioned that the real "meaning" (smType) of a component is only generated by observation in a larger correlation with other entities and can not be grounded in a symbolic description of the component instance alone. An indication of this fact would be a component instance that, though it has a constant behavior can have different meanings in two different observation contexts. For example the same camera instance that shows the entrance of a building additionally could show for one observer a certain street segment while for a third observer it shows the weather conditions, the water level of the nearby river and so on. Another example would be a temperature sensor on the outside of a package. It can mean the outside temperature (compared to the packages inside temperature) but also at the same time could have a meaning of inside temperature for the owner of a storage house the package is currently stored in.

2.4.3 Application subsystem

A context adaptive system built with the CAWAR framework typically consists of three subsystems: a) the adaptation subsystem embracing all parts that are responsible for adaptive behavior and which are subject to the frame problem b) the system environment including all service fulfilling components which are not permanently available due to resource restrictions and c) the application subsystem comprising a single system bootstrapper (System Seed) with all components vital to the running system. Each application usually has its own System Seed that can be installed and uninstalled separately. A System Seed Package typically includes:

1. A boot sensor,
2. An optional boot actuator and
3. The applications core system.

Usually the application core system initialized by the System Seed contains only a boot sensor specification and an administration component implementing that boot sensor. The administration component, usually a GUI, connects to the application origin server and from there downloads or updates a K-model XML file for the application and any necessary core system components that run in the domain of the application. These core system components are necessary for providing a required minimal functionality of the system. This may include at least the necessary framework components as well as a default context server, which handles the initial service communication by storing the messages to (persistent) context elements. Following the principles in section 2.4.1, this minimal functionality can of course be extended on the fly, in case the corresponding resources for fulfilling additional services becoming available.

3 Methodology for developing context adaptive systems

The development of context aware ubiquitous applications for realistic scenarios is today, fifteen years after the announcement of the vision of ubiquity by Mark Weiser [9], extremely difficult and, if at all, only prototypically possible. Development tools and methods are still in an early stage [6]. This is above all a consequence of the fact that these applications are very complex, e.g. regarding aspects such as multi-functionality, distribution and situational context. The development as well as the deployment of context adaptive systems is furthermore often associated with very specific challenges as changing environmental conditions and Unwanted Behavior [8].

The methodology proposed in this section considers these challenges and thereby states an iterative design methodology, starting with the design of adequate scenarios and closing with the final design of the adaptive system behavior of the considered application.

The overall methodology for designing adaptation models is structured into eight individual design steps, which in conjunction guide the engineering of the adaptation behavior of a context adaptive system. The particular design activities thereby chronologically build upon each other, so that certain results of previous activities are required by succeeding activities. However it is possible, and of course also recommended, to iterate through each individual phase if required. Figure 3.1 illustrates the particular steps of the proposed methodology.

It should be mentioned that the activities proposed in the methodology are not mandatory. But, since the proposition is based on several experiences gained from previous application development, we strongly recommend them. A brief description of the steps illustrated in figure 3.1 is given in the following sections.

One should also note, that the proposed steps are not intended to be a complete software engineering process or method. Instead these steps are merely expressing the differences in an idealized overall development process between context adaptive and non-adaptive systems. Thus it should be easy to integrate them in a customized version of any specific development method, creating an instance that is specially tailored for the development of context adaptive (parts of a) system. Furthermore, the methods described in this paper only focus on (late) requirements engineering related tasks and their transition into early design phases. Hence they are only the first step towards a bigger methodology for adaptation design in Ubiquitous Computing.

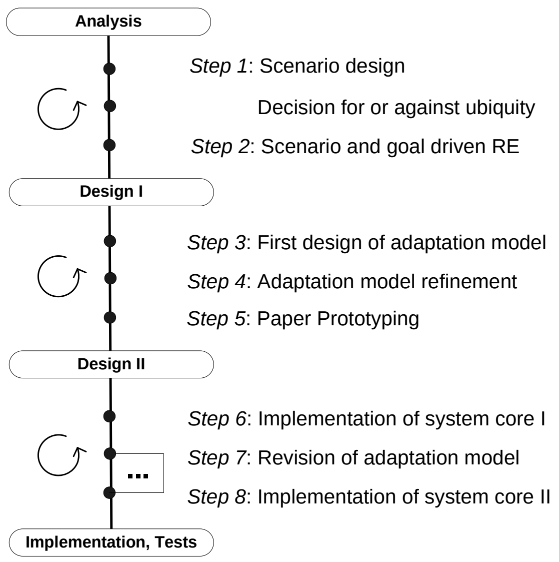

Analysis
Step 1: Scenario design
Decision for or against ubiquity
Step 2: Scenario and goal driven RE
Design I
Step 3: First design of adaptation model
Step 4: Adaptation model refinement
Step 5: Paper Prototyping
Design II
Step 6: Implementation of system core I
Step 7: Revision of adaptation model
...
Step 8: Implementation of system core II
Implementation, Tests

Fig. 3.1: The overall CAWAR design methodology

3.1 Step 1: Scenario design

At the beginning of any product development, one first of all needs to identify the required characteristics of the artifact under consideration. This is even more the case when developing software, since the value of software products heavily depend on their ability to satisfy the needs of their intended target group, e.g. users, administrators, etc. One obvious possibility for collecting user needs is to document the system usage in form of textual scenario descriptions. To facilitate the communication between all stakeholders such scenario descriptions should be formulated in a way, which is understandable both for end users and system designers.

Scenarios are typically used to clarify the purpose of the system and are situated at the very beginning of the design process, sometimes even in the course of marketing research. For systems operating in changing environments, such scenario descriptions should ideally contain first indications of relevant usage situations. The identification of situations is necessary for constructing an appropriate model of the usage context, which builds the basis for later adaptation decisions and therefore influences the entire system behavior. The fact, that the knowledge of domain experts is indispensable for designing adequate scenario models [16] should not be underestimated while designing scenarios. In the case that domain experts however are not at hand, some simple heuristics are proposed in section 3.1.2, which nevertheless should guarantee a somewhat systematically proceeding.

3.1.1 Term explanation

The notion of scenarios discussed in this paper is not equivalent to UML scenarios, since the latter are in fact instances of use cases, which already contain information about the aspired solution. We employ scenarios in a broader sense in order to express our first ideas of how the system under consideration will eventually be used. Thus in contrast to UML, we also allow scenarios, that do not directly aim at providing a certain solution.

A scenario in our understanding represents a concrete example for the usage of a considered system, whereas this system could be a virtual or an already existing one. It always has a fixed setting, which is explicitly introduced prior to the scenario description. Furthermore, every scenario is attended by at least one actor, i.e. a person, system, etc., which is trying to accomplish a task or reach a certain goal, respectively.

As already mentioned, the creation of scenario descriptions is a rather informal task, which ideally should be underpinned by the knowledge of domain experts and accomplished in close collaboration with the user. The following section discusses some details of the actual proceeding of designing scenarios.

3.1.2 Designing scenarios

We found that surprisingly few papers have concentrated on the topic of scenario design by now. Moreover it seems to be generally accepted, that creativity and domain knowledge are indispensable preconditions for designing adequate scenarios. We are trying by no means to belittle the contribution, which both creativity and domain knowledge obviously have on the design of appropriate scenarios as well as system specifications in general. And for sure no design methodology can impart the knowledge required for designing scenarios in arbitrary application domains. However, we recommend some generic design principles in the following, which have proven to be useful for systematically creating suitable scenario descriptions for ubiquitous systems.

- It is important to respect the constraints for scenarios mentioned earlier.
  - Scenarios are settled in a well defined context which is introduced at first.
  - At least one actor is attending the scenario, trying to solve a certain problem.
  - Scenarios describe a process composed by a set of actions and either describe normal situations or exceptional circumstances.
- From the multitude of all possible interactions between a certain actor and the system, fist of all those interactions are selected, which contribute to the most important functionalities of the considered system. This set

comprises especially those interactions relevant for the intended target group, i.e. users, administrators etc.

- One challenge when designing scenarios is to cover as many relevant usage situations as possible within a limited set of scenario descriptions. Consequently, descriptions of redundant functionality should be avoided in any case.
- Scenario design typically states an iterative process, in which each iteration results in a more elaborated set of scenario descriptions, until the latter is finally accepted by all involved stakeholders. Following questions may contribute the refinement of existing and the identification of additional scenarios.
  - Are all relevant functions covered by the current scenarios?
  - Are all user needs reflected by the current scenarios?
  - Is the context of a scenario described sufficiently accurate?
- If scenarios are written in plain text, it is helpful to structure existing scenarios into individual sections. Each section thereby concentrates on at most one system function or user need.

In case experts with a deeper understanding of the considered problem domain are at hand, some additional methods can be applied in order to collect the necessary information for subsequent analysis and design activities. Depending on the corresponding activities for collecting these information, the methods are divided into *prompting techniques*, *observation techniques* and *creativity methods*. We outline an exemplary method for each category below:

Prompting techniques

One challenge of all prompting techniques is to *prompt* the *right people* with the *right questions*. Interviews, questionnaires and checklists or the so called *On-Site-Customer* [17] are typical methods contained in this category group. A discussion concerning different prompting techniques alone, as analyzed in [18], is far beyond the scope of this paper. Instead we exemplary present some basic principles of such techniques by means of *direct questions*, which are asked throughout almost any interview.

Direct questions are thereby used to eliminate ambiguities, which may be contained in informal scenario description. Beside selecting a representative group of users for being questioned about the considered system, the order in which the certain questions appear is crucial for an appropriate outcome of the interviews. A simple concept for prompting the questions in order of their importance is to use a *decision trees* as depicted in figure 3.2. By means of direct questions the decision tree is traversed from the root illustrating the first vague idea to a certain leave containing one possible solution for the considered problem. The traversal thereby constrains the set of possible solution with every question. For that reason a wrong decision due to an inaccurate question at the beginning of the tree leads to a solution, which is far away from what the interviewee actually wants.

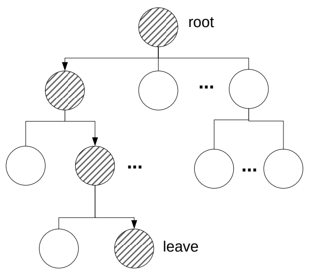


Fig. 3.2: Decision tree with one possible solution variant

The decision tree should be created in close collaboration with the interviewee, which iteratively reviews the current tree concerning the accordance with his own mental model. This again should minimize the occurrence of ambiguities, which may result from inaccurate formulations during the interview. Following questions may serve for giving the interview the intended direction and help to identify the general objectives, a certain user is trying to achieve with the system. The questions are thereby formulated from the viewpoint of a user:

- For what does users employ the system?
- Which aspects are important during system usage?
- How does the system ease their life?

It should be mentioned that interviews alone are not sufficient for resolving all ambiguities which typically occur during system analysis [19]. Interviews always should be combined with complementary techniques such as observations shortly discussed subsequently.

Observation techniques

Due to the fact that users usually have problems with expressing what they actually want, it is a good idea to circumvent this communication difficulties by observing them while doing their duty. *Contextual enquiry* is a proceeding typically applied in the context of usability testing, which helps to gain a better understanding of the user, his workplace and operational environment as well as her typical tasks, problems and preferences [20]. A contextual enquiry session is always held in the familiar environment of the observed user, whereby one or several persons observe the user in typical usage situations of the considered system. The observations are documented and analyzed afterwards. This document in turn can be used for the creation of further scenario descriptions. Contextual enquiry is of course constrained to the analysis of already deployed systems to be augmented or replaced, respectively. However this proceeding cannot be applied to gather information about fictive systems and preliminary prototypes. In such cases the following methods may reveal the desired information.

Creativity methods

Creativity methods as *Brainstorming* or *Mind Mapping* [21]are used for systematically collecting ideas and for enhancing the creativity of the involved persons. We exemplary outline the concept of Mind Mapping, whereas an overview and a detailed description of several creativity techniques can be found in [22]. The idea behind Mind Mapping is to annotate examined artifacts in a structured graphical manner, which also illustrates the relations between individual ideas, thus facilitating the learning process of human beings. Mind Mapping thereby regards the circumstances, that humans actually both think and learn in a non-linear way, i.e. triggered by perceivable keywords, new associations and structures are permanently evoked within the brain. The concept moreover unburdens users of handling empty phrases, so that the concentration on associative keywords is facilitated. The graphical notation furthermore eases the exposition of "mental gaps", i.e. immature reasonings that need additional reflections. Within a Mind Map such a gap attracts the user's attention the same way as a table containing an empty row catches someone's eye.

3.1.3 Conditions for ubiquity

As soon as an initial set of scenarios, which reflects the usage of the considered application, has been designed and was accepted by all involved stakeholders, a decision concerning a ubiquitous realization should be discussed on basis of this scenario set.

Figure 3.3 illustrates a simple checklist that is applied in order to determine whether a ubiquitous realization of the application is recommendable or not. Since a ubiquitous realization is associated with additional expenses, the decision should be carefully considered. In case this decision cannot be determined on basis of the previous domain knowledge, this step also can be delayed until the requirements engineering has been completed (step 2 in the overall design process), which ideally results in a complete and non-divergent set of situational requirements sufficient for unambiguously resolving the ubiquity decision.

If this decision reveals no need for ubiquity, the further system design can be continued with any conventional software engineering process. Otherwise step 2 will be applied afterwards, namely a scenario and goal driven requirements engineering. The accomplishment of this first step should result in "adequate scenarios" describing the usage of the considered system.

From this description the first step in the CAWAR methodology does not seem to differ much from classical scenario based requirements engineering approaches. However it should be pointed out, that the main difference is not just finding enough scenarios that reflect all system requirements as complete as possible.

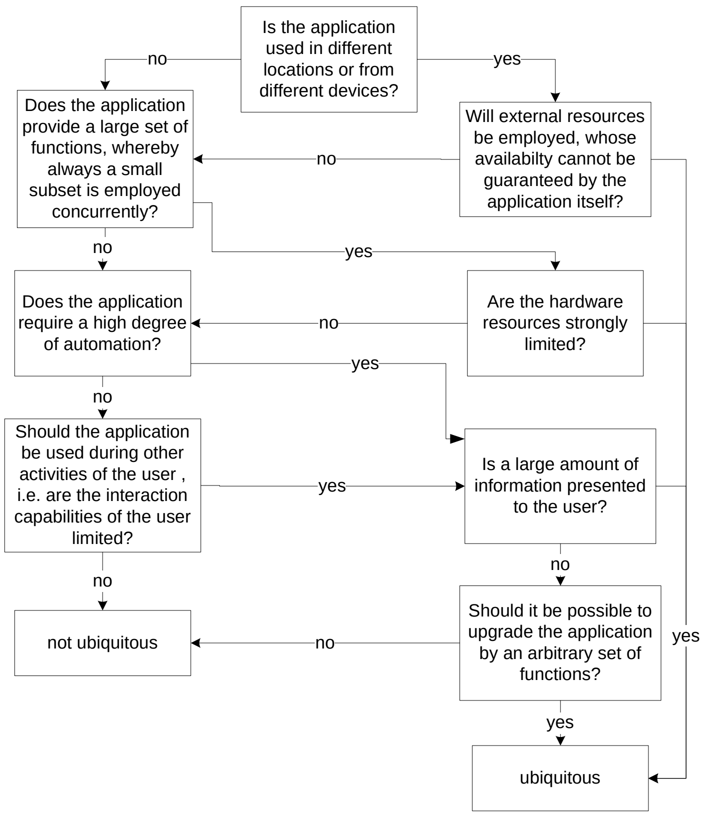

Is the application used in different locations or from different devices?
no
yes
Does the application provide a large set of functions, whereby always a small subset is employed concurrently?
Will external resources be employed, whose availabilty cannot be guaranteed by the application itself?
no
no
yes
Does the application require a high degree of automation?
Are the hardware resources strongly limited?
no
yes
no
Should the application be used during other activities of the user , i.e. are the interaction capabilities of the user limited?
yes
Is a large amount of information presented to the user?
no
no
Should it be possible to upgrade the application by an arbitrary set of functions?
not ubiquitous
no
yes
yes
ubiquitous

Fig. 3.3: Decision for or against ubiquity and hence explicit adaptivity

Instead, this first step is concerned with finding a set of scenarios, that is as complete as possible *regarding all conceivable situations*, the system could exist in. Furthermore, these situation scenarios are often reused as a direct input for the later system model. Consequently they are not discarded, after the corresponding requirements have been derived from them.

3.1.4 Reflection and outcome

Boehm [23] concluded that 54% of all errors ever detected in software projects studied at TRW were in fact detected *after* the coding and unit testing activities. 45% of these errors were attributable to requirements and design activities, whereby only 9% were attributable to coding activities [24]. Such inquiries in our opinion reveal the need for a methodical and iterative approach towards the analysis of complex software systems, combined with an early integration of user feedback and additional tool support. However, we experienced that currently no methodology exists, which guides the systematical design of adequate usage scenarios without delegating the crucial parts to domain experts. Despite this circumstance, we tried to present some useful concepts together with generic heuristics, which provide a reasonably accurate process concerning the design of scenarios – even in the absence of domain experts.

The outcome of the activities described in step 1 is a sufficiently complex scenario description of the application under consideration. In most cases this step also reveals the decision, whether a ubiquitous realization of the application is recommended and hence an explicit handling of context adaptation is necessary.

3.2 Step 2: Scenario-based requirements engineering

On basis of scenarios derived from the previous step, a textual-based requirements elicitation is conducted subsequently. The main purpose of this elicitation is to derive an adequate set of requirements, which accurately reflects the intended system behavior and is as consistent and complete as possible. In case of ubiquitous and context adaptive systems, that are likely to be employed within highly dynamic environments characterized by changing conditions and corresponding user needs, it is indispensable to collect as many information as possible regarding the individual situations, in which the identified requirements occur. As defined in section 2.1, such perceivable information are referred to as *context*.

The universe of requirements and perceivable context information makes up the main content of the requirements specification, which describes the intended behavior of the considered system and states the basis for later system models as well as their corresponding realizations. This specification often inheres the status of a binding contract, in which both principal and system developer agree upon the specifics of the system under construction.

In literature many approaches can be found, which deal with the accomplishment and the (partial) automation of requirement elicitation. However, most approaches focus on the application of very specific techniques, which are not adequate for all types of considered applications and projects. Other models are formally founded but are extraordinary complex, thus making them only feasible for system aspects that are highly safety critical. This observation of course strengthens the assumption, that currently no “one size fits all” method exists. In order to take remedial action for the domain of Ubiquitous Computing, we consequently extract some generic and promising concepts of previous approaches, that can easily be integrated into the recommended Cawar development process for ubiquitous applications. Again it should be mentioned, that the presented concepts are merely recommendations: the choice for one or two concepts certainly depends on personal preferences as well as project specifics and hence is ultimately delegated to system analysts.

3.2.1 Linguistic analysis

According to [25], an analyst must use every possible information source available, in order to define conceptual models of the problem domain and to deduce requirements, respectively. These information sources may inter alia comprise very large documents written in natural language. In contrast to techniques regarding the interactive elicitation of requirements, the analysis of complex textual documents has been mainly disregarded by now. Analysis concepts and tools as described in [26] and [27], are in particular motivated by providing support for RE-analysis of large textual documents.

Among such tools supporting the identification of requirements, the probabilistic tools (probabilistic NLP - Natural Language Processing) have proven to be more robust than simple rule based approaches. Instead of trying to understand the analyzed documents, these tools rather try to extract interesting attributes contained within the documents. Thus on basis of often recurring keywords as *shall*, *must*, *should* or *will*, the document is augmented by semantical tags, which enable a categorization of captured text passages according to domain specific categories like telecommunication, finance, etc.

Perhaps it should be emphasized at this point, that even if such analysis tools support analysts in extracting relevant information from given documents, they cannot accurately deduce the requirements implied by the underlying documents *on their own*. The requirements extracted by these tools indeed will always remain incomplete and never form a precise snapshot of the actually involved system requirements [27]. This circumstance implies, that the ascertained information is strongly limited by the quality of the underlying source documents. In consequence, the following sections provide some recommendations for the manual elicitation of requirements, which can be applied complementary to automatic tools in order to enhance the quality of the gathered requirements.

3.2.2 Heuristics

Beside a deeper understanding of the application domain, the heuristics outlined in the following can help to identify the requirements as well as the associated conditions under which these requirements are valid, i.e. their situation context. We assume that the heuristics are applied to textual scenario descriptions as they result from the previous step 1 in the design methodology. This is however not a real limitation, since at least all informal models can be equally expressed by means of textual descriptions. As mentioned earlier the quality of the analyzed documents thereby plays a major role, since we assume that the heuristics can only reveal those requirements, which were previously committed to paper. Implicit assumptions contained in such documents obviously cannot be captured by simple text analyses. We recommend the following heuristics for the identification of requirements and their associated context:

- Since a certain scenario always involves at least one acting person (see section 3.1), all systems and persons interacting with the considered system should be identified as possible *actors*, which are annotated as nouns and typically participate in certain actions.
- *Actions* are the essence of a requirement and describe the desired system behavior. They are indicated by means of predicates, which are responsible for the statement of an analyzed sentence. Typical examples taken from

the case study described in section 4 are actions like *display*, *remind*, *enter*, *delete*, *record* etc.

- In the case of ubiquitous and context adaptive systems, actions are usually taking place within a certain *context*, which characterizes the accurate conditions or circumstances of the considered action. Some exemplary keywords that indicate a contained context information within a sentence are *during*, *as*, *in*, *on*, *because* etc.
- The *objects* involved in a certain action may expose indications for artifacts like data or components, which in turn are relevant for an identified requirement and are subject to a certain action, respectively.
- *Attributes* as well as *adjectives* in the context of actions and objects typically indicate, how a considered action should be accomplished. Moreover they may relate to additional context information, which must be considered in the resulting K-Model of the system as described in section 2.2. The quality of an identified action is an example for such a context information. Attributes as well as adjectives can alternatively be specified in form of non-functional requirements, if this seems more intuitive.

3.2.3 Requirements elicitation

As soon as the first requirements and their associated context information were identified, they should be documented in a structured and concise manner. The gathered requirements are technically often quite unspecific and therefore cannot directly be implemented by developers. These rather abstract requirements are also referred to as *goals*. In the context of goal driven requirements engineering the concept of goals can be defined as follows:

> *A goal is defined as something, that a stakeholder hopes to achieve in the future.*

The concept of goals thereby has several advantages: while on the one hand goals help to identify further requirements as described in [28], they on the other hand justify the existence of certain requirements, in that a requirement is necessary for accomplishing an associated and more abstract parent goal. Hence a considered set of requirements is complete, as soon as the fulfillment of each individual requirement also implies the fulfillment of a superior goal [29]. In addition, goals are more stable than requirements due to their higher degree of abstraction [30]. To emphasize this one can say, that requirements are related to goals the same way as programs are related to design specifications: requirements namely realize their underlying goals.

3.2.4 Linking requirements and scenarios

For the purpose of documenting a gathered set of goals/requirements, RE approaches like CREWS [31] inter alia recommend to link goals/requirements with scenarios. This enables an easy lookup in order to find all goals/requirements which were deduced from a certain scenario. Hence every goal/requirement is attached with a justification in form of the associated scenario description. The concept of *requirement chunks (RC)* thereby provides a concise notation for representing the linked goal/requirement-scenario pairs. A certain requirement chunk always consists of a concrete goal or requirement and the associated scenario extract, from which the goal or requirement was elicited.

As already mentioned in previous sections, the requirement concerning a context adaptive system usually relate to a certain context which characterizes the conditions or situations, under which the requirement is valid. In order to reflect this circumstance, the concept of requirement chunks is consequently augmented by an additional *context* information. Hence every individual chunk consists of a goal or requirement, the associated context in which the former is valid as well as the scenario description, both goal/requirement and context are deduced from. Table 3.1 illustrates such an exemplary requirement chunk taken from the case study of the context aware task scheduler Cats.

G1: The user wants to be notified *silently* about incoming messages.	C1: The user is currently attending a meeting.	S1: While Jeff is currently attending his 8 o'clock meeting, the scheduler reminds him silently of a consecutively appointment in order to not disturb the participants of the ongoing meeting.

Tab. 3.1: Augmented requirement chunk containing a situation depending goal

3.2.5 Elaboration of goals

As soon as an initial set of goals and contexts have been extracted from the scenario descriptions, the identified goals can be refined by means of a simple notation. *Goal refinement graphs* represent an intuitive graphical notation proposed in [32], which inter alia help to systematically deduce technical requirements from a set of abstract goals. An exemplary goal refinement graph originating from the domain of a train control system modeled in [32] is illustrated in figure 3.4.

The overall process of constructing a goal refinement graph can be outlined as follows: at first the identified goals are denoted with a concise and meaningful name, which summarizes the essence of a depicted goal. Subsequently a goal hierarchy is elaborated by means of the graph: abstract goals are systematically *refined* into subgoals, which are necessary in order to achieve their parent goal. As indicated in figure 3.4, the refinement is driven by asking *how* a certain goal can be achieved by more concrete subgoals (top-down approach).

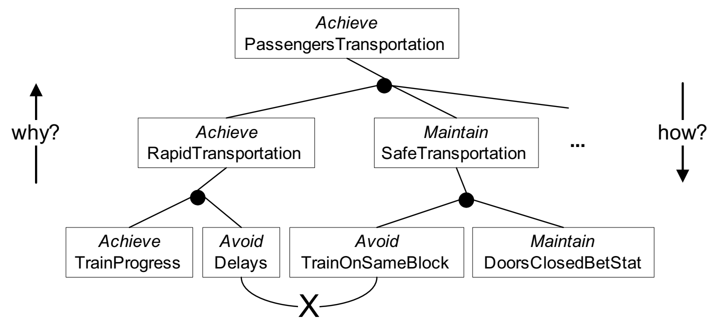


Fig. 3.4: Goal refinement graph concerning a train control system

This refinement process is iterated, until the resulting goals are accurate and concrete enough for being technically implemented. These goals form the leaves of the goal refinement graph and are referred to as *requirements*. In case such a requirement cannot be accomplished by means of a software component, it is called an *assumption*.

On the other hand do questions investigating *why* a certain goal was identified, result in the identification of additional goals, which provide the reasoning for the considered goal (bottom up approach). A parent goal may be linked with it's individual sub goals via logical *OR* or *AND* relations. An AND-relation thereby indicates, that *all* sub goals must contribute to the fulfillment of the parent goal, whereas an OR-relation only requires the fulfillment of *at least one* sub goal. As the two requirements *AvoidDelays* and *AvoidTrainOnSameBlock* in figure 3.4 indicate, may conflicting goals be identified and annotated while elaborating a goal refinement graph. Depending on the implications imposed by two conflicting goals, the conflict should be resolved subsequently, e.g. by elaborating another goal refinement, which results in a consistent set of requirements.

The Language Kaos [33] [30] provides the possibility of formalizing the assumptions and requirements within a goal graph by means of a temporal logic. Such formalizations are of course associated with additional expenses and therefore should be carefully considered from an economical point of view. However, when dealing with safety critical requirements, it might be mandatory or at least beneficial to formalize the requirements, in order to use automated model checkers which may reveal the presence of conflicting requirements. In case conflicting requirements are identified, several possibilities might be considered for resolving them: an alternative refinement of the graph may for instance result in another consistent set of goals. Another possibility is to avoid constraints or other external circumstances that are responsible for the conflict. In some cases it might be necessary to weaken conflicting requirements or to completely remove them, respectively. Other strategies for resolving conflicts can be found in [33].

3.2.6 Reflection and outcome

This section outlined several methods for eliciting requirements from different information sources in the context of the considered application. It was mentioned several times, that the knowledge of domain experts is essential for accomplishing a successful requirements analysis. Beside a deeper understanding of the application domain, the introduced heuristics can be applied in order to guide the analysis in a systematical way for both domain experts and non-experts.

The notion of goals as used in goal oriented approaches has proven to be very practical for the identification, elicitation, elaboration and validation of system requirements. However it should be mentioned, that scenario based and goal oriented approaches are only one possibility among many others, we recommend for methodically gathering the requirements of a system under construction. It again pretty much depends on the personal experiences and preferences, which approach to choose for an individual case.

As a result of this design step a sufficient number of requirements should be identified together with their corresponding validity conditions represented as context. Due to the notion of requirement chunks and goal refinement graphs, these requirements are already documented in a concise and structured way, which enables the tracing back to underlying analysis documents as the scenario descriptions resulting from step 1. On basis of these requirements and contexts an initial context adaptation model is subsequently designed, which explicitly describes the adaptive behavior of the system under construction. Since the Cawar methodology represents an iterative approach towards the design of context adaptive systems, the requirements elicited at this stage do not have to be complete. A further elaboration of the requirements namely is conducted by means of an adaptation model in step 4 of the design process.

3.3 Step 3: First adaptation model

Starting from a set of requirements and associated context data, an initial adaptation model of the considered system can be constructed. Using the requirement chunks introduced in the previous section facilitates the process, although it is not mandatory to document requirements in this way. The purpose of this activity is to express the identified requirements by means of the modeling technique described in section 2.2, i.e. by means of the K-Model. In the following we recommend a couple of design rules, which help to express the textually documented requirements by means of the four service types *sensors*, *context elements*, *interpreters* and *actuators*.

As soon as the first services of the initial K-Model are identified, there are in principle three slightly different ways, how an initial K-Model can be further elaborated. Which way to use in the individual case thereby mainly depends on the designer's point of view as well as already existing components whose services should be considered in the adaptation model.

3.3.1 Transformation of requirements

It was argued in section 2.3, that it is strongly recommended to explicitly model those parts of the system behavior, that are likely to change during the application life cycle and hence are prone to Unwanted Behavior and the frame problem, respectively. In order to circumvent this issue, the services exposing the affected behavior are transparently specified by means of a calibrateable context adaptation model. Due to its technical realization on the basis of the Cawar framework, this K-Model can be modified (calibrated) on demand, in case UB is recognized.

A question typically arising from the described circumstances is, how an adaptation model revealing the intended system behavior can be systematically constructed. As each adaptation model consists of only four basic service types (sensors, interpreters, context elements and actuators), the requirement chunks resulting from previous design steps somehow have to be *translated* into such an appropriate model representation. The challenge of this translation mainly consists of expressing the requirements and context data in the light of a possibly unfamiliar modeling technique. Beside a good portion of sure instinct, the following heuristics should facilitate a systematical approach towards the design of adaption models. We thereby illustrate the general proceeding by means of a simple example and assume that the requirement chunk depicted in table 3.2 should be systematically translated into an initial K-Model extract.

Requirement	*Context*	*Scenario*
R1: Situation dependant notification of users.	C1: 1. User attends meeting 2. Notification matures 3. User is able to explicitly interact with the system.	Sc1: User John is currently attending a meeting while the cancellation of a subsequent appointment is announced. Because this information is rated as important for John, the system employs the vibration alarm for notifying him silently in order to not disturb other participants.

Tab. 3.2: Requirement chunk as starting point for constructing a K-Model extract

3.3.2 Heuristics

The following rules help to identify the initial elements of a K-Model and are beneficial for transforming an existing requirements catalogue:

- Validity conditions which are contained in the second column of the requirement chunk are initially expressed by means of *interpretations*. To clarify that an interpretation represents an identified usage situation of the system, it is annotated as a *situation adaptor* (circle). At first an identified situation is merely specified by additional comments (see figure 3.5).

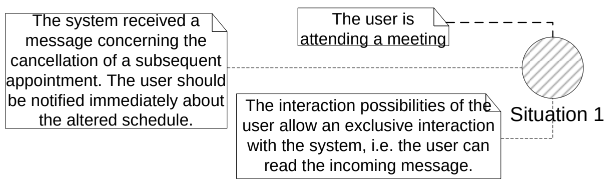

The system received a message concerning the cancellation of a subsequent appointment. The user should be notified immediately about the altered schedule.
The user is attending a meeting
The interaction possibilities of the user allow an exclusive interaction with the system, i.e. the user can read the incoming message.
Situation 1

Fig. 3.5: Identifying and commenting situations

- A requirement is initially expressed by an *adaptation action* (rounded rectangle), which is connected with the corresponding situation adaptor symbolizing the validity conditions of the considered requirement. The adaptation action thereby respectively represents an objective or nominal condition, which should be implemented by an appropriate *actuator*. The underlying rationale for this modeling approach is to express the following causal dependency: *if* the situation depicted by the situation adaptor occurs, *then* execute the requirement described by the adaptation action. To clarify the purpose of an identified adaptation action, it also can be annotated by explanatory comments (omitted in figure 3.6).

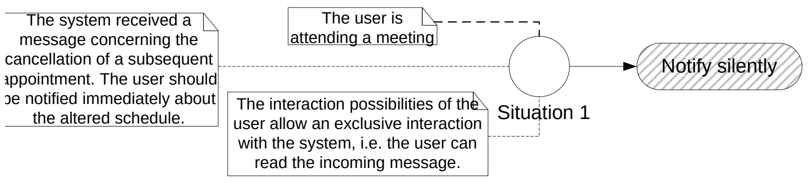

The system received a message concerning the cancellation of a subsequent appointment. The user should be notified immediately about the altered schedule.
The user is attending a meeting
Notify silently
The interaction possibilities of the user allow an exclusive interaction with the system, i.e. the user can read the incoming message.
Situation 1

Fig. 3.6: Relating situations and actions

- In case a requirement chunk exposes certain information, which are relevant for achieving the considered requirement and are somehow measurable or can be derived from measurable information, then this information should be formulated as *context* (parallelogram). The second column of a certain requirement chunk as well as the comments concerning the situations depicted in an adaptation model are good candidates for such context information, which are appropriately integrated into the model as shown in figure 3.7.

3.3.3 Missing design patterns

Needless to say that beside the mentioned heuristics also a deeper understanding of the expressive power of the considered modeling technique is advantageous when designing adaptation models. The design of K-Models reveals a couple of

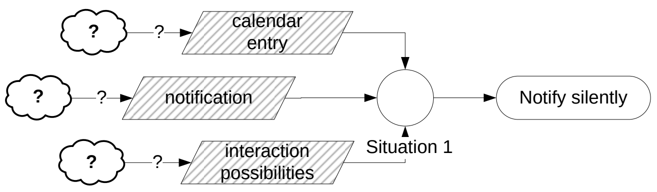


Fig. 3.7: Identifying and integrating context information

similarities to the design of object oriented models: with increasing modeling experience designers will possibly detect an elegant solution for often recurring problems, which are ideally documented in form of design patterns [34]. Since the formulation of design patterns usually requires an empirically relevant degree of experience with the considered modeling techniques, no design patterns have been invented so far, which truly facilitates the design of context adaptive systems. Exceptions are some specific templates concerning technical adaptation actions and premature patterns for expressing technically oriented concepts for monitoring and redundancy on basis of adaptation models.

With respect to the lack of mature patterns we therefore recommend a trial and error approach for designing adaptation models. Due to their simple structure and the loose coupling between involved services, different design alternatives can easily be established and modified within a certain K-Model. The calibration mechanism also contributes to and eases the handling of the fluid nature of adaptation models, in that it allows for their modification at runtime in order to adapt the system behavior to changing and unforeseen user needs and environmental conditions[1]. Accordingly one can say that K-Models are not at all written in stone and can be arbitrary modified and discarded without great effort.

3.3.4 Preliminary design

As soon as the first elements of the K-Model are written down, these initial model fragments can easily be extended and elaborated by applying some simple rules. At this stage in the design, the previous model is usually still full of gaps. We therefore recommend three approaches in order to fill these gaps and to enlarge the model fragments to a coherent adaptation model. These approaches are based on different points of view concerning the modeled system behavior and mainly differ in the specific model extract, which states the starting point for further model enlargements. Hence it depends on the premises of the considered system and the preferences of the designer which approach to focus on in the individual case.

[1]In fact the modular composition of system behavior by means of decoupled services which already may be available at design time suggests and supports a rapid prototyping of K-Models [35]

Situation-oriented approach

This approach starts from the individual *situations* which form the validity conditions of the identified system requirements. Accordingly, all situations which are relevant with respect to the usage of the system are first of all expressed by means of special interpreters called situation adaptors. Among the three approaches, the situation-oriented is probably the most intuitive one, since the notion of situations used in adaptation models abstract from technical details and corresponds to the commonly used notion of situations.

Starting from an individual situation there are again two possibilities to expand the adaptation model. The first one explores all information that are required for characterizing a depicted situation, i.e. how these information can be measured via sensors or derived from measurable information via interpreters, respectively. We assume that measurable information comprise physically measurable information as temperature and speed as well as measurable information in the broader sense, i.e. inputs made by users or other systems over direct or indirect input channels (keyboard, microphone, infrared sensor, etc.). This strategy enlarges the adaptation model towards the set of sensors and their subsequent sensor context, respectively.

Alternatively one can initially examine, how the application should behave in a certain situation, i.e. which requirements the application typically needs to fulfill in this situation. The realization of a situation dependent requirement is first of all expressed by an *action*, which the application triggers whenever the considered situation occurs. This strategy results in an elaboration of the model towards the actuators and their precursory adaptation context.

Sensor-oriented approach

This approach mainly focuses on examining all information, which can be measured by actual components of the system core and the environment, respectively. In contrast to the situation-oriented approach discussed in the previous section, this strategy is rather technically oriented and is concerned with identifying the actual information sources available. As available sensors are the starting point for this approach, it is especially useful in case the system or it's environment already provides a relative fixed set of sensors (as e.g. in a car). Thus designers are typically concerned with the question of how to use the information gathered by sensors for specifying the intended behavior of the overall system.

Another advantage of the sensor-oriented approach is, that the model is enlarged according to the flow of data within the model and the process of context adaptation described in figure 2.2, which mainly consists of the three steps (a) information acquisition, (b) situation identification and (c) implementation of adaptation decision.

Consequently the individual steps within this approach are the following:

1. Identify all available sensors for gathering the information relevant for the context adaptation. The context contained in the requirements catalogue should indicate which sensors to integrate.

2. Connect a context element which each sensor, which decouples the sensor from other services in the model. This decoupling via buffers helps to avoid data loss in case of unreliable and error-prone sensors and facilitates their transparent exchange.

3. Combine the gathered sensor context appropriately by means of interpreters, which help to identify certain usage situations and drive the inference of adaptation decision, respectively[2]. The design of an adequate decision logic by means of interpreter networks is probably the most challenging aspect of the modeling process.

4. Once a certain usage situation as well as the associated requirements are identified by the decision logic developed in step 3, these requirements are realized by means of actuators, which propagate the system reaction or the calculated adaptation decision to the executing system components. This decision mostly results in a changed (observable) system behavior in consequence of a context triggered data manipulation or a system reconfiguration.

Adaptation-context-oriented approach

This third approach assumes that the intended adaptations of the system behavior are at least partially known in advance. In other words, this approach is suitable in case a future state or objective, which specifies the situation-dependent behavior of the system is already known. Those adaptations may result from changed user needs or a changed resource availableness, which must be handled in order to fulfill certain (non-) functional requirements. Such a requirement is expressed in form of an adaptation context, which contains the necessary information for the upcoming adaptation of the system. The requirement is afterwards read by an actuator, which actually implements the contained adaptation decision by interacting with the system core or the environment. Since the necessary decisions concerning a certain adaptation may be of a very technical nature – especially if the adaptation aims at adhering some functionality in cases of component failure – this again states a rather technical approach.

If the information concerning the necessary adaptation is presently rather unspecific in terms of technical aspects, it is also possible to express the adaptation decision by means of an *adaptation action*, which will be refined later on in the iterative development process.

[2]This correlates to an abstraction of information

Recommendations

Independent of the individual approaches we recommend developing a K-Model along a coherent *thread*, whereas a thread is meant to be a chain of certain elements in the adaptation model, which according to the three steps of the adaptation process (fig. 2.3) realize a certain functionality of the system under construction. This practice facilitates the concentration on a certain aspect of the overall system behavior and encourages the creation of modular and coherent K-Models.

Whenever some aspects of the system behavior have already been modeled as thread fragments, it is often advantageous to use a combination of the previous described approaches. One decides for each individual thread, how to enlarge its margins according to the three approaches (see fig. 3.8). How to fill the remaining gaps with intermediate context in order to connect the individual threads is discussed in later sections.

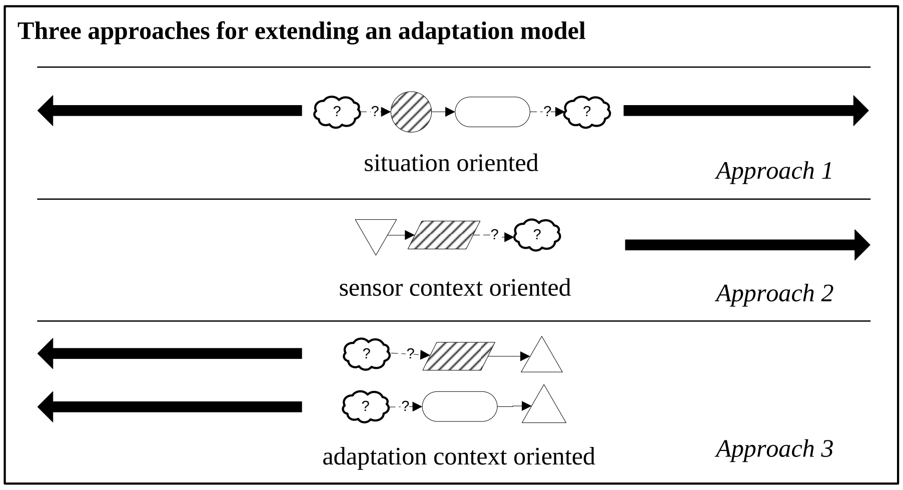

Three approaches for extending an adaptation model
?
situation oriented
Approach 1
sensor context oriented
Approach 2
adaptation context oriented
Approach 3

Fig. 3.8: Three approaches for extending an adaptation model

3.3.5 Reflection and outcome

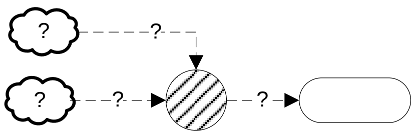
The integration of requirements in a first preliminary version of the K-Model is usually one of the most challenging steps in the development process, since a good portion of experience in designing adequate adaptation models is crucial for this activity. Generic rules for assisting unexperienced designers are difficult to define, while most requirements and usage situations are application specific. Design patterns as described in [34] would help to overcome this issue. However, such patterns can hardly consolidate without being used and validated in a variety of system development projects. At the moment only four case studies were designed using this approach.

Due to the comfortable graphical notation of the K-Model consisting of only four service types, such models are easy to create and revise. The outcome of this step is a first preliminary K-Model. This model needs by no means to be complete and may also contain adaptation actions and situation adaptors. Both elements enable a more comfortable modeling of the system, since also vague information of the analysis, which are not specified in detail by now, can be integrated into the model. Both elements however have to be refined later on.

3.4 Step 4: Context requirements engineering

After step 3 an initial adaptation model is available, which usually does not yet cover all aspects of the considered system behavior. Hence the purpose of this step is to fill in the missing elements, thus completing the adaptation model by preferably expressing all requirements and context data identified in the requirements specification. The graphical notation of K-Models facilitates the exposure of gaps previously hidden within the textual requirements specification.

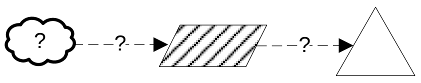
The refinement and elaboration activities described in the following are in principle applied to all elements contained in an adaptation model. However, isolated or unconnected elements which represent the “open ends” of a depicted adaptation thread are of particular interest within this step.



Situation adaptors

The following questions are useful for identifying further requirements concerning situation adaptors:

- Which information is necessary for identifying a pictured situation in an unambiguously manner?
- Are there any further requirements/actions/needs that apply to this situation?
- Is it possible to further decompose the associated requirements?
- Do any of the associated requirements occur in further depicted situations?
- Do the existing situations cover all relevant usage situations, or are there distinguishable situations not considered so far?
- Is the correlation between the contexts used for a situation detection and the requirements applying to this situation comprehensible? If not, additional interpretation chains should be appended, which make the considered adaptation thread more readable.



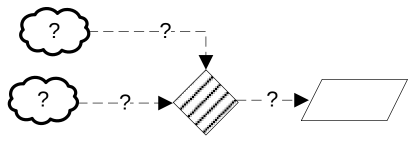
Intermediate context

The composition rules for adaptation models require sensors to write into exactly one subsequent context element and actuators to read from at least one context. Interpreters in turn read from at least one input context and write to at least one output context. Thus, the sole existence of sensors, interpreters and actuators already determines a good portion of the required contexts. The remaining question is, how the necessary intermediate context can be methodically identified.

Intermediate context comprises all contexts that are not *directly* connected with sensors, situation adaptors or actuators. It represents those contexts, that are necessary for deriving an information relevant for the adaptation, which cannot be expressed by means of sensor context, situation context and adaptation context. Intermediate context is for example necessary for multiplexing the contexts measured by multiple sensor or for translating distinct units (km/h vs. mph).

The following questions are useful for identifying further requirements concerning a depicted context element:

- Can a depicted sensor context be directly used in order to reason about an actual situation, or is additional information necessary?
- Is a sensor context appropriate for making an adaptation decision, or is further information necessary?
- From which information is a depicted situation or adaptation context composed of? How can this information be deduced from existing contexts?



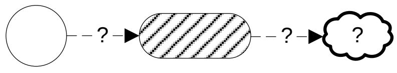
Interpreters

The following questions are useful for identifying further requirements concerning a depicted interpreter:

- Can a necessary information be supplied by an existing context element? If not, does a combination of existing contexts provide the necessary information?
- How does a depicted sensor context contribute to an adaptation decision?
- Is it possible to enhance the readability of an adaptation model by appending further interpretation threads? If so, then enhance the model.
- Are several contexts relevant for making an adaptation decision? If so, they should be bundled by an interpretation chain and delivered to the responsible actuator afterwards.

It should be mentioned that the investigation for interpreters and the search for intermediate contexts cannot be regarded as isolated activities: intermediate context always results from interpretation of other contexts, while an additional interpreter often produces or requires further intermediate contexts. With this in mind, the considered activities rather complement each other.

Adaptation actions



Adaptation actions as well as situation adaptors are auxiliary concepts for facilitating an "intuitive" modeling of requirements and their associated validity conditions in form of context. The intention of this notion is to express causal relations of the system behavior in an understandable way. Such causal relations can be modeled by simple *if–then*-dependencies between a situation and its corresponding action, thereby abstracting from technical details and modeling constraints.

An adaptation action expresses the (possibly vague) notion of an action, which should be triggered in case the associated situation occurs – one might say a context triggered action. During the design process such adaptation actions must be decomposed by means of the conventional model elements available for adaptation models – in contrast to situation adaptors, which already represent specific interpreters. Subsequent to the Prototyping accomplished in step 5, a K-Model must not contain any adaptation actions. Otherwise this services may not be implemented by means of the underlying CAWAR framework. In the simplest case, an adaptation action is transformed into a single context element, which is directly read by an actuator implementing the specified action. If complex actions need to be carried out, it may be necessary to substitute the adaptation action by an additional interpretation chain.

The following questions are useful for both identifying further requirements concerning a depicted adaptation action and indications for their possible transformations, respectively:

- Shall a depicted action also be triggered in further situations?
- Which steps are necessary for implementing the action? Should the action be decomposed in a set of (smaller) sub-actions?
- Can an action be directly transformed into an adaptation context, or are further interpretations necessary for this task?

3.4.1 Reflection and outcome

The activities described in the previous section help completing the initial K-Model until most of the requirements are modeled, i.e. all identified system functions are covered by the adaptation model. Some simple engineering rules may contribute during this activity.

Related work that also concentrates on the derivation of intermediate context (cf. [36]), should be mentioned at this point. It is often assumed, that the necessary context information for making an adaptation decision are already known, i.e. the system's future states and objectives are certain in advance. In order to derive further necessary context, [36] recommends the usage of a modification of the *Activity Theory* known from psychology. However, providing support for determining the appropriate content of a context as well as the question of how to combine different contexts in order to derive a desired information, remains an open issue.

Application of metrics

There are in principle two metrics which enable an evaluation of an adaptation model (cf. [4]). An evaluation by means of these metrics may expose some fundamental deficiencies within the considered model. However, in order to produce an expressive evaluation, the considered K-Model must satisfy certain characteristics, which in some cases cannot be assumed a priori. The adaptation metric for instance counts the number of situation adaptors within the model for evaluating the system's ability to adapt to different situations of usage. If for any reason the designer deliberately renounces to use situation adaptors within a certain model (design decision), the metric will certify a non-adaptive model, even if the system may differentiate between several situations very well. Similar constraints apply for the balancing metric, since it builds upon the adaptation metric. We therefore recommend the usage of such metrics with great care and preferably only after subsequent iterations of the design process, when informations concerning the system models have consolidated.

What to model explicit?

An important question concerning the refinement of the K-Model is, which aspects of the system behavior should be explicitly modeled within the adaptation model, and which aspects are encapsulated within the individual services of the model. The obvious answer is, that every aspect which is subject to the frame problem should be modeled explicitly. However, it depends on the designer, the application domain and certain other external factors, which aspects are prone to the frame problem. As a rule of thumb, every service that is prone to change or reconfiguration requests, should be modeled explicitly. Similarly, every aspect that is somehow related to personalization should be modeled explicitly. From a technical point of view, every service that possibly can be realized by external components not contained within the system core (e.g. the mobile device), should also be explicitly modeled in the adaptation model.

The outcome of this step is a refinement/extension of the initial adaptation model. At this stage of maturation, the K-Model should ideally cover all functionalities of the desired system. However, there still might be some obscurities about certain aspects of the system, technical realizations or even coherences of aspects. Such obscurities are treated in subsequent steps and iterations, thus weaving new insights into the model.

3.5 Step 5: Prototyping

Prototyping is an appropriate concept for gaining new insights and early feedback concerning the system under construction. Prototypes may implement certain aspects of a system in order to study different technologies, functionalities or even parts of the system model. Therefore, prototyping in combination with software testing serves as an evaluation tool for interfaces (mainly user interfaces) and for providing deeper insights concerning the usability of the considered system. For instance, a prototype may be used to test, if the input/output behavior of some component is correct or if an algorithm fulfills the specified (non-)functional requirements.

In the case of context adaptive applications, one might also be interested in studying the correlation of certain contexts, their relation to identified situations and their effects on the adaptations of the system. An appropriate prototype may highly contribute to resolving obscurities within a K-Model. In the following, we concentrate on two different variants of prototypes, namely paper prototypes and executable prototypes, and discuss when to use which one.

3.5.1 Paper Prototyping

Paper prototypes are probably most cost-efficient and provide a good opportunity to gain early feedback from users and other stakeholders. They are mainly used for evaluating the usability concerns of (graphical) user interfaces. In addition to this conventional usage, Paper Prototyping (PP) may also contribute to enhancing a considered adaptation model. A detailed description of paper prototypes can be found in [37]. In the following, we sketch out those aspects of PP, which are of particular interest when evaluating adaptation models.

Each service within the adaptation model must eventually be represented within the paper prototype. In contrast to user interfaces considered in conventional interactive systems, context adaptive systems also require an investigation of the interfaces responsible for context acquisition, situation detection and decision making known from the context adaptation process. A paper prototype therefore consists of a set of the four basic elements (sensors, interpreters, contexts, actuators) realizing this context adaptation process. The characteristic information of each element is thereby annotated on a single page of paper, which has the following content:

Context elements

are information carriers within the system and inter alia characterize relevant information gathered from the environment. Hence the content of each individual context element is most interesting for the paper prototype. This content is for example captured in terms of key-value pairs.

Interpreters
: are information processing units. Besides their in- and outputs in form of context, the underlying algorithms used for processing this information is of particular interest. The associated rules or algorithms are typically denoted as pseudo-code.

Sensors
: are the data sources of each K-Model. Some sensors receive direct user input, while others acquire environmental information (indirect inputs). Because each sensor stores the received input into one subsequent sensor context, sensors are not necessarily included within a paper prototype. However, if required the sensors can be integrated as comments, annotating the corresponding sensor context.

Actuators
: constitute the information sinks and executive elements within an adaptation model. Analogous to interpreters, the underlying rules for deploying the adaptation decisions are the most relevant aspects of actuators, which are denoted in terms of pseudo-code.

The communication between associated elements in the adaptation model is then optimized on the basis of this PP. This examination gains a particular importance, if already existing components should be integrated in the adaptation model. In case a commercial component providing some sophisticated functionality should be integrated in an adaptation model as an interpreter, the associated elements of this interpreter should be optimized to match the interface and constraints inherent to this component, to achieve an optimal collaboration between the involved elements. Another advantage of Paper Prototyping is, that it narrows the gap between design and realization of an adaptation model and supplies useful details for the subsequent implementation of individual elements.

In the course of a Paper Prototyping session, several roles are defined for simulating the usage of the system under consideration. These rules are shortly described in the following, whereby the *context* role is an extension of the role model sketched out in [37]. This context reflects the (indirect) acquisition of contextual information as opposed to direct user input.

User
: The user interacts with a paper based version of the considered user interface. She is guided by the moderator in order to carry out realistic tasks with this paper mock-up. The tasks performed by the user should involve critical system aspects, which decide on the acceptance of the system.

Computer
: One of the system designers plays the role of the computer. The computer thereby simulates the interface behavior as well as all other system aspects

requiring a system reaction. However, she does not provide any assistance concerning the accomplishment of the postulated tasks.

Moderator
The moderator has the function of guiding the user. She defines the user's tasks and provides assistance when necessary.

Developer
Developers observe the interactions between the user and the computer. Their main task consists in making notes of observable problems and the usability of the simulated system.

Context
The moderator or an additional person takes over the part of the system context and simulates the indirect inputs concerning the current circumstances of the execution.

However, sometimes paper prototyping lacks the capability to properly reproduce certain dynamic aspects of context adaptive systems like their "automatic intelligence". More sophisticated methods such as Wizard of Oz based techniques [38] perform better in some sense, but they also come at some cost. There is some promising research going on in that area, which should be shortly described in the following section.

3.5.2 Executable prototypes

Executable prototypes do in many aspects enable a more sophisticated evaluation than purely paper based mock-ups. There currently exists a rapid prototyping environment for the CAWAR Methodology (CAWAR Prototyping Environment or shortly CPE [35]) which leverages the concept of context adaptation to evolve the same prototype over the different development phases – from a scenario controlled slide show into a full blown interactive prototype that could demonstrate the application on real devices. In other words, CPE enables the development of a reconfigurable prototype, which better reflects the actual needs and insights of the current development stage. Simulations are replaced by actual components in a flexible and stepwise fashion as described in the following.

In an initial version of the CPE prototype, all components like context sensors and application actuators are mapped onto simulation components, that do not really calculate, sense or act. At the first stage, this is equivalent to having a tool that supports creating electronic paper prototypes. Most services are simulations and map predefined stages in a scenario to drawn screen-shots of devices with the expected output. This is particular useful, if during the development phase real sensors and actuators are not available or too expensive for early deployment.

In a next step, real decision making logic can be implemented to allow an interactive evaluation of the system. Further steps can replace several simulated services with real implementations and devices up to the point where it is possible to make usability tests or run automatic sample scenarios in conjunction with testing framework instrumentation on real hardware. Afterward we would, step by step, replace the remaining simulation components by real ones while simultaneously modifying the adaptation behavior, just as the current phase of development would require. At first we would do this in order to evaluate existing and to identify new user requirements, respectively. Later, certain design alternatives are evaluated against each other and finally all CPE instrumentation is removed, thus releasing the system as a final product. The CPE tool thereby fully supports the K-Model including its adaptation and calibration capabilities. During the whole process at any time a change of the adaptation model is possible to better suite to new, changed or wrongly interpreted requirements only requiring at least corresponding simulation components being present to allow for evaluation of the changed model.

3.5.3 Application of prototypes

The decision concerning the appropriate prototype mainly depends on the current stage in the development process and the purpose of the prototype itself. In the early phases of the development cycle, when the system model typically underlies frequent change, we recommend the usage of paper based mock-ups describing the intended system behavior. In order to examine certain K-Model services in more detail and to gain the first feedback, such mock-ups can easily be sketched and revised without great effort. Thus, paper prototypes are a cost-efficient alternative for a simple presentation and evaluation of preliminary system designs, respectively. However, paper mock-ups are often insufficient for reasoning about alternative design decisions concerning performance or scalability aspects of the considered system.

If on the other hand the adaptation model has reached a certain stage of maturation, i.e. the model gradually becomes stable, designers are typically interested in evaluating certain design decisions concerning architectural solutions, functional behavior or the designed adaptation logic of the modeled system. In such cases, it is usually a good idea to develop an initial executable prototype as described for the Cawar Prototyping Environment. Such a prototype can be iteratively extended, until the adaptation behavior is ready to be deployed in an environment, where production stable versions of the service implementations are available. This narrows the gap between system design and implementation, leading to a seamless development and evaluation of context adaptive systems.

3.6 Step 6: Implementing the system core

Due to the development activities carried out in previous steps, a mature adaptation model reflecting all identified requirements and associated contexts is available. The prototypes of step 5 are important indicators for the upcoming implementation of all services included in the K-Model. At first, a decision is made, which of the specified services required by the adaptation model should be realized by components within the system core, and which of them should be moved to the system environment, respectively. Heuristics for guiding this decision are given in the subsequent section.

However, the implementation of each individual component itself is accomplished by conventional concepts of software and systems engineering. The actual implementation is therefore beyond the scope of the Cawar methodology. The only precondition the service fulfilling components must ensure, is to implement the framework specific interfaces for sensors, contexts, interpreters or actuators, respectively. In case the interface cannot be directly implemented due to 3rd party software components, these components must be bound to the framework via intermediary wrapper components (Adapter pattern see [34]) fulfilling the interface imposed by the framework. A short discussion concerning which services should explicitly appear within an adaptation model at all, and which functional behavior should be implicitly encapsulated within a component, respectively, is given in section 3.4.1.

Internal vs. external realization

An external implementation of a service (e.g. as a web service) should be preferred, if the corresponding functionality is not permanently required and moreover directly accesses other external resources (e.g. network connectivity) which can not be made available in any situation of usage and can not be decoupled via context elements. Does the service in contrast only require the infrastructure provided by the core system, and its functionality is needed in almost every situation, an internal implementation of that service should be considered.

Services that offer functionality in situation dependent quality and complexity must be considered separately. For example a display service hosted at some pocket PC. The context adaptive system provides this display service permanently with a minimal quality dependent on the resolution of the pocket PC. As soon as the user enters his office, the system discovers an appropriate monitor with a higher resolution and automatically binds this monitor to the display service. As this example illustrates, such services require a component within the system core for permanently providing a required minimal functionality. Components temporarily providing additional or enhanced functionality are implemented externally, whereby the Cawar framework automatically handles the context adaptation of the system by reconfiguring the involved services. For enabling such reconfigurations, the framework infrastructure must be permanently available. Accordingly, it is implemented within the system core as well.

3.6.1 Reflection and outcome

The initial structuring of the overall system functionality into the system core and its environment is mainly motivated by the lessons learned during the prototyping accomplished in step 5. The decisions are based on the availability of resources and the dependencies between functions. The implementation of certain services may require the addition of further adaptations, resulting in an extension of the adaptation model. This issue is discussed in more detail within the subsequent step. Moreover, the core components implemented during this step should be used for replacing the according simulation components within the associated CPE prototype.

3.7 Step 7: Revision of adaptation design

To enhance the quality of the context adaptive system under construction, and hence of the underlying adaptation model, the Cawar methodology requires a certain degree of iterative development [39]. After all services contained in the adaptation model have been implemented as internal or external components, the model usually needs to be iteratively updated. The reason for this revision is, that during the implementation of services often details concerning a better modeling of certain adaptations become visible, which can be conducted in this step. Such modifications result in a modified adaptation model, which in general is augmented by additional, usually more technically oriented adaptations (e.g. discovery of external components, optimizations, autonomic failure recovery etc.). The resulting adaptation model furthermore provides the right stage of maturation for additional evaluations concerning consistency and the appliance of appropriate metrics, respectively. The outlined modifications are described in more detail within the following sections.

3.7.1 Adding secondary adaptations

As already mentioned in section 3.6, the components of context adaptive systems may be realized within the system core or its environment, respectively. Sometimes even services within the system core may depend on other services which are realized externally. Such *external service dependencies* emerge, if a certain service can not provide its functionality without the usage of an external resource, e.g. an external device (printer, screen) that is installed at a certain location within the system environment. Such an external resource in turn must be bound to a corresponding service within the system. In case several alternatives for the external resource are available, and moreover the selection of the appropriate alternative is situation dependent (cost, location, quality etc.), the decision concerning the resource selection should be explicitly modeled within the adaptation model.

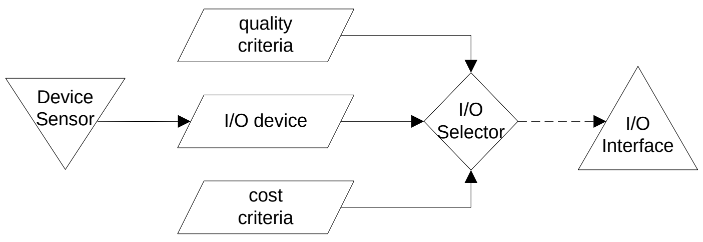


Fig. 3.9: Secondary adaptation of the attending case study

An example for such a resource selection modeled by means of an adaptation model is shown in figure 3.9. The example originates from the attending case study of section 4 and can be summarized as follows. The interpreter *I/O-Selector* decides on basis of certain situation dependent criteria (*quality criteria*, *cost criteria*) and the currently available displays in the system environment (*I/O-Device*), which device (i.e. component) to use for realizing the *I/O-Interface* service. If no external displays are available, the standard display of the pocket PC is used for realizing the *I/O-Interface* service.

Technically, the selection of appropriate components is achieved by modifying a dedicated *meta context* element that is associated with the *I/O-Interface* service. Such meta context elements are inherent features of all service types and inter alia comprise syntactic (SyType) and semantic (SmType) service descriptions, which are used for binding the service to appropriate service fulfilling components. Possible components fulfilling the services of a K-Model are identified during a runtime discovery (e.g. UPnP, UDDI). In case such a component matches with the description of a service contained in the K-Model, this component is bound to the according service. The dashed line within figure 3.9 is a shortcut for expressing that *I/O-Selector* is able to modify the service description (meta context) of *I/O-Interface* and thus affects *I/O-Interface*'s reconfiguration, i.e. which component should be bound to *I/O-Interface*. This shortcut avoids representing meta contexts within graphical K-Models.

Marking secondary adaptations

The secondary adaptations added after the first implementation of the services contained in the core system usually originate from technical oriented considerations, resulting in *technical context* and associated services as depicted in fig. 3.9. Heuristics for identifying such technical context can be found in [4]. If such technical context and associated interpreters, sensors etc. appear within an adaptation model, they should be explicitly marked as shown in fig. 3.10. These markings indicate, that those services must be realized within the system core in order to avoid further technical context, that would otherwise result from an external realization of such services.

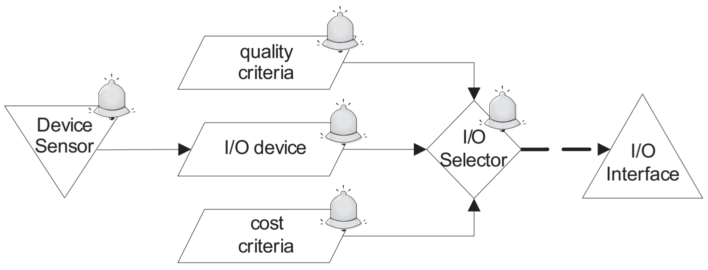


Fig. 3.10: Markup of services involved in secondary adaptations

Further adaptations, consistency and metrics

In addition to the technical oriented adaptations described above, any enhancements and modifications identified during the implementation of the core system should be integrated into the adaptation model during this development step. These modifications often base on new insights acquired during the core implementation. Sometimes, certain design decisions have to be revised after actually checking them out within the running system.

This stage of the development process is also perfect for applying consistency checks for the adaptation model as described in [4]. If not already done, the adaptation model should now be stable enough for the application of metrics as outlined in section 3.4.1 and introduced in [4]. If carefully applied, such metrics help to analyze certain characteristics as the adaptivity potential of the considered application. In the course of this step, the improvements resulting from these analyses are integrated together with the modifications caused by the emergence of technical context.

3.7.2 Reflection and outcome

The revision of the adaptation design aims at resolving external service dependencies that potentially emerge from the technical implementation of the core system. Additionally, necessary modifications identified in prototype analyses as well as any other modifications to the adaptation model are incorporated during this step. Further analyses of the adaptation model according to metrics allow for revealing further inconsistencies or flaws.

The outcome of this step is a revised K-Model, which includes all secondary adaptations that are necessary due to the implementation of services. The marking of services involved in secondary adaptations is an important input for the subsequent step. In a second iteration, the remaining core system is revised and implemented. Marked services thereby must be implemented in the core system to avoid the emergence of further technical context and associated external service dependencies, respectively.

3.8 Step 8: Revision of system core

In this final step all new services of the modified adaptation model are implemented as internal or external components. The required activities are analog to that already described in step 6: (a) decision for internal or external realization and (b) conventional implementation of components. The only exception is, that all services involved in the technically oriented adaptations added in previous step 7 should be realized *internally*. Otherwise an external implementation of these services can cause single points of failures, which derive from possible external service dependencies.

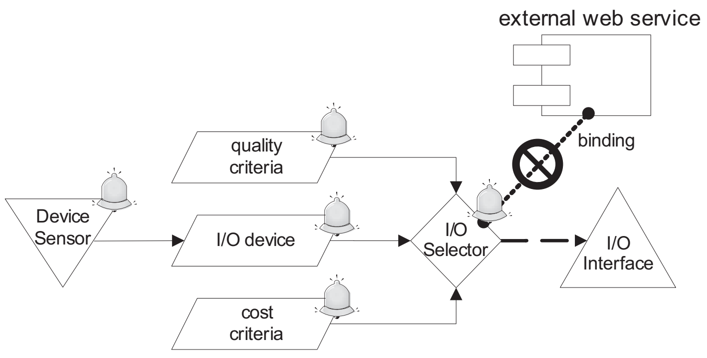


Fig. 3.11: Situation dependent selection of a component is realized internally

Figure 3.11 illustrates the avoidance of further external service dependencies when implementing the remaining services of the adaptation model. For all services concerned with technical context, the decision for their service implementation (internal vs. external) is anticipated. The depicted interpreter *I/O-Selector* decides about which (external) component is bound to the *I/O-Interface* service. If *I/O-Selector* itself is implemented by some external component, in turn another service is necessary for deciding which external component should be bound to the *I/O-Selector* service, which itself however is required to make the same decision for the service *I/O-Interface*. In order to break this chain of service dependencies, a service (like *I/O-Selector*) deciding about the binding of another service to a possibly external resource, is not allowed to be implemented by an external component itself. Hence *I/O-Selector* and any other marked service must be implemented internally, i.e. permanently bound to a component of the core system.

3.8.1 Reflection and outcome

The revision of the system core is the final step in the overall CAWAR methodology. This step produces implementations for all services contained in the adaptation model, which were chosen to be implemented within the system core. Moreover, for all services to be externally realized, we assume the existence of appropriate service fulfilling components. As a result of this step, an augmented prototype is available, in which all simulated components are replaced by actual components of the core system and its environment, respectively.

3.9 A word on iterations

From a practical point of view, iterations within the development process are a promising approach for refining and elaborating the system behavior. This is even more true for context adaptive systems, which have to consider a large number of different usage situations, users, functionalities and their relationship, respectively. Therefore, the prototypes resulting from early iterations of the development cycle usually differ from the users' expectations. Consequently it is often necessary, to reenter the development process in order to narrow the gap between users' expectations and the provided system functionality.

With this in mind, the CAWAR methodology was designed as an iterative development approach. Generally, CAWAR recommends two coarse grained development cycles (step 1 to 5 and step 6 to 8, see fig.3.1), whereby each individual step can be iterated as well. As soon as a new prototype results from the activities until step 5, it typically occurs that new aspects are identified or certain deficiencies are revealed due to user feedback. At which stage to reenter the development process in order to integrate the gained insights, mainly depends on the type of user feedback during the prototyping. Sometimes system functionalities are well considered, but their mapping onto usage situations is inappropriate. In this case the adaptation model probably should be revised within step 4. However, it may also occur that completely new usage scenarios are identified. This would of course require a revision of the whole development process, starting with the design of the new usage scenarios within step 1.

The second development cycle (step 6 to 8) assumes, that the intended system behavior and the associated adaptation logic is relatively stable. It therefore focuses on rather technical aspects concerned with the appropriate implementation of the adaptation design and the actual system functionality[3]. It is also possible, that new aspects identified in the second loop induce a reentry into the first loop – however this is not the average case.

[3]One might say *primary* system functionalities as opposed to the functionalities realizing the context adaptation

4 Proof of concept

4.1 CATS – The Context Aware Task Scheduler

As mentioned in the introduction of this paper, a prototype system was developed for evaluating the theoretical concepts introduced in the previous sections. The application described in the following is some kind of personal assistant named CATS (**C**ontext **A**ware **T**ask **S**cheduler), which uses different context information for organizing the user's daily schedules as helpful as possible. Appointments, events and other personal tasks of a user can thereby be imported into the application context from different sources such as email, online calendar or even public event repositories. The CATS furthermore provides some basic functionality for managing this data. Several forms of adaptive notification functionalities were implemented, which for instance inform about rearranged or conflicting schedules and remind users of upcoming appointments. The type of a notification is context aware, i.e. the application for instance realizes, if the user is currently participating in a meeting and in consequence informs him unobtrusively or not at all, if the notification is rated of low importance. Besides the functionality of manually rearranging schedules or setting filter rules concerning unwanted events gathered from public repositories, a mechanism for the automatic rearrangement of timely overlapping events and appointments is provided. In combination, these features save users a good portion of consistency checks and enable a contemporary reaction to canceled or just announced appointments.

4.2 Designing CATS by means of the CAWAR methodology

The CATS was modeled using the CAWAR methodology defined in section 3. Starting with the design of a simplified usage scenario, which is restricted to one functional aspect of the overall CATS system, the individual design activities proposed in this methodology as well as the corresponding work products resulting from each activity are described on the basis of an illustrative example.

4.2.1 Scenario design

The overall process begins with the creation of textual documents, which describe the usage of the system as well as its intended behavior. The scenario extract below was identified during an early analysis phase of the case study and covers a representative situation of how the prototype should work:

11:30 a.m. Mr. Williams is currently participating in an important conference. Thanks to the wireless LAN connection in the conference room, his pocket PC receives a message, indicating that the meeting announced for 5:00 p.m. on that day was canceled. On basis of the registered schedules for today, the CATS client assumes that Mr. Williams at the moment is attending a meeting. To avoid unnecessary interruptions of participants, he is either notified silently in the form of a textual message or notified delayed, in case the message is rated of low importance for him. The CATS client afterwards automatically resets the canceled meeting and furthermore informs Mr. Williams, that due to this new situation he is now able to re-attend his weekly squash lesson for 6:30 p.m.

4.2.2 Extracting user goals, requirements and context

Once a sufficiently complex scenario of the considered system was constructed, user needs as well as (situation-dependent) requirements should be extracted on basis of this description. In case the desired information cannot be identified immediately, since the scenario for example comprises several hidden assumptions, complementary modeling techniques like UML Use Cases can be useful for revealing this hidden assumptions, hence making them explicit. Collecting *user goals* thereby helps to identify the general objectives, users want to achieve with the aid of the system. An analysis of the scenario extract depicted above could for example reveal the following user goals:

- Missing of arbitrary dates should be avoided in either case
- Important appointments (e.g. meetings, theater) should not be interrupted.
- Mr. Williams needs to react on unforeseen changes in his schedule in a timely manner

User goals form another beneficial source for gathering requirements and moreover provide a rationale for the latter, i.e. the elicitation of a certain requirement can be justified, since it contributes to the completion of the corresponding and more abstract user goal. Consequently, user goals are usually too abstract for a direct derivation of implementation decisions. However, they are useful for elaborating and completing the requirements catalogue of the considered system. Picking up our Cats example, the requirements depicted in table 4.1 could (amongst others) be derived from previous results.

We strongly recommend to document each identified requirement along with its optional validity condition in terms of *context* as well as the underlying scenario extract in form of a requirement chunk [40], which offers a concise description for specifying requirements and facilitates the tracing between scenarios and requirements.

Requirement	**Context**	**Scenario/Description**
R1: Communicate events, appointments, emails, messages	n/a	Events, appointments, emails and miscellaneous messages as well as notifications should be communicated to the user on demand.
R2: The user wants to be notified unobtrusively.	The user is currently attending a meeting.	1. The client realizes a canceled appointment. 2. The client tries to determine the current situation of usage. 3. Due to a current meeting rated as important, the user is notified silently.

Tab. 4.1: Modeling CATS by means of requirement chunks

4.2.3 Construction of an adaptation model

As soon as an adequate collection of requirements and contextual data was elicited, an initial K-Model can be constructed on basis of this collection. The purpose is to express both requirements and context by means of the modeling possibilities supported by adaptation models. The proceeding thereby can be summarized as follows:

- Initially, the associated context of a discrete requirement can be expressed in terms of a specific interpreter called *situation adaptor* (circle in fig. 4.1), representing an identified usage situation. Textual comments are used for specifying this situation.
- The requirement itself is symbolized as an *action* element (rounded rectangle in fig. 4.1), representing a nominal condition or target state, which should be realized by a subsequent context adaptation. The association between situations and actions thereby signals the following: *if* the participating situation occurs, *then* the associated action is performed, thus implementing the underlying requirement.
- All context information, which are necessary for unambiguously identifying a pictured usage situation and furthermore can be measured or somehow derived from measurable information, respectively, should be modeled as *context* elements (parallelograms in fig. 4.1). Heuristics for identifying the different types of context data (sensor, situation, adaptation, intermediate, technical) can be found in section 3.3, 3.4 and 3.7, respectively.

Figure 4.1 illustrates one possible solution for expressing the identified requirement R2 described in the requirement chunk above by means of an exemplary K-Model extract, consisting of three *context elements*, one *situation adaptor* and one *adaptation action*. Once the first elements of an adaptation model were identified, they can be augmented by means of the approaches drafted in fig. 3.8.

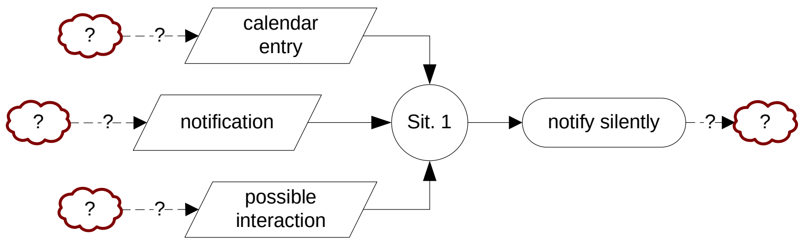


Fig. 4.1: Constructing an initial K-Model

In case a depicted *thread* of elements cannot be augmented at the moment, since for example the necessary information concerning this thread and its specific function is not available by now, the next requirement chunk from the underlying specification should be modeled and integrated into the present K-Model. This step is repeated until all requirement chunks are expressed on the basis of sensors, context, interpreters and actuators. As soon as the resulting K-Model has reached a certain stage of maturation, a paper prototype of the model is designed, which provides the right abstraction level for examining each individual element as well as the collaboration of associated elements in more detail. We refer to [41] for a detailed description of the individual development activities as well as the entire modeling of the CATS prototype.

5 Conclusion and outlook

The generic process proposed in this paper enables the systematical design of adaptive, context-aware applications in the emerging field of Ubiquitous Computing. The process consists of eight fundamental steps, providing assistance for relevant analysis and design activities. It concentrates on the elaboration and specification of the adaptation behavior rather than discussing implementation details. The activities result in the generation of an adaptation model of the considered system, as well as useful implementation instructions for application specific components. In conjunction with the underlying CAWAR framework, both adaptation model and the necessary components together form a complete and executable adaptive system, whose transparent behavior can be easily communicated to the user. Furthermore, based on the concept of *calibration*, the adaptation behavior is moreover totally reconfigurable as demonstrated in [4], and therefore can be adjusted anytime to fit changed circumstances, even if these embrace a changed mental model of the user.

Further research efforts in this context are directed towards *tool support* for the concise notation of graphical adaptation models, which particularly should support several techniques for defining model cuttings. Other considerations concern an (semi-) automatic transformation between graphical adaptation models and equivalent XML specifications, comparable to diagram based code generators provided by modern development environments. Except for some very specific, technically oriented or domain dependent patterns, no explicit *design patterns* have been previously proposed, which provide a reusable solution for common design problems appearing in adaptation models. Generic design and modeling guidelines for context adaptation and a concise requirements engineering methodology for the development of adaptive systems are still important research topics. Further considerations regard the tracing between requirements and adaptation models and the definition of metrics respectively, which allow evaluations of arbitrary adaptation models on the basis of a few model characteristics.

List of Figures

Bibliography

[1] Gerhard Fischer. User Modeling in Human–Computer Interaction. *User Modeling and User-Adapted Interaction*, 11(1-2):65–86, 2001.

[2] A. Sutcliffe, S. Fickas, and M. Sohlberg. PC-RE: a Method for Personal and Contextual Requirements Engineering with some Experience. *Requirements Engineering*, Vol. 11(No. 4):157–173, 2006.

[3] Anind K. Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, College of Computing, Georgia Institute of Technology, 2000.

[4] Michael Robert Fahrmair. *Kalibrierbare Kontextadaption für Ubiquitous Computing*. PhD thesis, Department of Informatics, Technische Universität München, Germany, 2 2005.

[5] Albrecht Schmidt. *Ubiquitous Computing - Computing in Context*. PhD thesis, Computing Department, Lancaster University, U.K., 2002.

[6] Nigel Davies, James Landay, Scott Hudson, and Albrecht Schmidt. Rapid Prototyping for Ubiquitous Computing. *Pervasive Computing*, 4(4), 2005.

[7] L. Kolos-Mazuryk, P. A. T. van Eck, and R. J. Wieringa. A survey of requirements engineering methods for pervasive services. Deliverable TI/RS/2006/018, Freeband A-MUSE, Enschede, 2006.

[8] Michael Fahrmair, Wassiou Sitou, and Bernd Spanfelner. Unwanted behavior and its impact on adaptive systems in ubiquitous computing. In *ABIS 2006: 14th Workshop on Adaptivity and User Modeling in Interactive Systems*, Hildesheim, Germany, October 2006.

[9] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 3(265):94–104, September 1991.

[10] B. Rhodes. The wearable remembrance agent: A system for augmented memory. *Personal Technologies Journal Special Issue on Wearable Computing*, 1:218–224, 1997.

[11] Jens Wohltorf, Richard Cissée, Andreas Rieger, and Heiko Scheunemann. BerlinTainment: An Agent-Based Serviceware Framework for Context-Aware Services. In *MobiSys Workshop on Context Awareness*, 2004.

[12] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces and Refinement*. Springer-Verlag, 2001.

[13] Elaine Rich and Kevin Knight. *Artificial Intelligence*, chapter The Frame Problem. Internation Edition, second edition, 1991.

[14] E. Mohyeldin, M. Dillinger, M. Fahrmair, W. Sitou, and P. Dornbusch. A Generic Framework for Negotiations and Trading in Context Aware Radio. In *Software Defined Radio Technical Conference – SDR-TC*, 2004.

[15] Eiman Mohyeldin, Michael Fahrmair, Wassiou Sitou, and Bernd Spanfelner. A Generic Framework for Context Aware and Adaptation Behaviour of Reconfigurable Systems. In *16th IEEE Intl. Symposium on Personal In-door and Mobile Radio Communications – PIMRC*, 2005.

[16] Eric Kemp-Benedict. Narrative–led and indicator–driven scenario development. a methodology for constructing scenarios. Technical report, February 2006.

[17] Juha Koskela and Pekka Abrahamsson. On-site customer in an xp project: Empirical results from a case study. In *EuroSPI*, pages 1–11, 2004.

[18] G. Browne and M. Rogich. An empirical investigation of user requirements elicitation: Comparing the effectiveness of prompting techniques. *Jounral of Management Information Systems.*, 17(4):223–250, 2001.

[19] Donald C. Gause and Gerald M. Weinberg. *Exploring Requirements: Quality Before Design*. Dorset House Publishing Co., Inc., New York, NY, USA, 1989.

[20] Hugh Beyer and Karen Holtzblatt. *Contextual design: defining customer-centered systems*. Morgan Kaufmann Publishers Inc., 1998.

[21] Tony Buzan and Barry Buzan. *The Mind Map Book*. BBC Books, 2003.

[22] C. Malorny, W. Schwarz, and H. Backerra. *Die sieben Kreativitätswerkzeuge K7*. Hanser München/Wien, 1997.

[23] B. W. Boehm, R. K. McClean, and D. B. Urfrig. Some experience with automated aids to the design of large-scale reliable software. In *Proceedings of the international conference on Reliable software*, pages 105–113, New York, NY, USA, 1975. ACM.

[24] Alan M. Davis. *Software requirements: analysis and specification*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1990.

[25] Keith A. Butler, Chris Esposito, and Ron Hebron. Connecting the design of software to the design of work. *Commun. ACM*, 42(1):38–46, 1999.

[26] Leonid Kof. *Text Analysis for Requirements Engineering*. PhD thesis, Technische Universität München, 2005.

[27] P. Rayson, R. Garside, and P. Sawyer. Assisting requirements engineering with semantic document analysis. In *RIAO 2000: Recherche d'Informations Assistie par Ordinateur, Computer-Assisted Information Retrieval Intl. Conf.*, pages 1363–1371, Paris, France, April 2000. Collège de France.

[28] D.T. Ross and K.E. Schoman. Structured analysis for requirements definition. *IEEE Transactions on Software Engineering*, 3(1):6–15, 1977.

[29] K. Yue. What does it mean to say that a specification is complete? In *4th Intl. Workshop on Software Specification and Design*, Monterey, 1987.

[30] Axel van Lamsweerde. Building formal requirements models for reliable software. In *Reliable Software Technologies: 6th Ada-Europe Intl. Conf.*, 2001.

[31] Anne Wolter. *Seminar Requirements Engineering: CREWS – Scenario Based Requirements Engineering.* Universität Duisburg–Essen, 2005.

[32] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *5th IEEE Intl. Symposium on Requirements Engineering*, page 249. IEEE Computer Society, 2001.

[33] Axel van Lamsweerde, Emmanual Letier, and Robert Darimont. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. Software Engineering*, 24(11):908–926, 1998.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: abstraction and reuse of object-oriented design.* Springer-Verlag, 2002.

[35] Michael Fahrmair, Wassiou Sitou, and Bernd Spanfelner. Seamless Development and Evaluation of Context Adaptive Systems. In *UBICOMP - USE 2007: 9th International Conference on Ubiquitous Computing- Workshop on Ubiquitous Systems Evaluation*, 2007.

[36] Manasawee Kaenampornpan and Eamonn O'Neill. Modelling context: An activity theory approach. In *EUSAI*, pages 367–374, 2004.

[37] Carolyn Snyder. *Paper Prototyping - The fast and easy way to design and refine user interfaces.* Morgan Kaufmann, 2003.

[38] Steven Dow, Blair MacIntyre, Jaemin Lee, Christopher Oezbek, Jay David Bolter, and Maribeth Gandy. Wizard of oz support throughout an iterative design process. *IEEE Pervasive Computing*, 4(4):18–26, 2005.

[39] V.R. Basili and A.J. Turner. Iterative enhancement: A practical technique for software development. *IEEE Trans. Software Eng.*, SE-1(4):390–396, 1975.

[40] M. Tawbi, F. Velez, C. Souveyet, and C. Achour. Evaluating the crews-l'ecritoire requirements elicitation process. In *4th IEEE Intl. Conf. on Requirements Engineering*, 2000.

[41] Christian Leuxner. Adaptation design in ubiquitous computing. Diploma thesis, Technische Universität München, August 2006.