The Human in the Loop:  
User Participation in Self-Adaptive Software

Dissertation
in partial fulfilment of the requirements for the degree

Doktor der Naturwissenschaften (Dr. rer. nat.)

for submission to the

Faculty of Electrical Engineering and Computer Science
University of Kassel, Germany

March 2014
M. Sc. Christoph Evers

Advisers:
Prof. Dr. Kurt Geihs, University of Kassel
Prof. Dr. Albrecht Schmidt, University of Stuttgart

Additional Examination Committee Members:
Prof. Dr.-Ing. Ludger Schmidt, University of Kassel
Prof. Dr. Gerd Stumme, University of Kassel

Date of Defense: 24 July 2014
Contents

List of Figures v
List of Tables vii
List of Listings ix
List of Algorithms xi
List of Abbreviations xiii
Abstract xv
Acknowledgements xvii

I Foundations 1

1 Introduction 3
1.1 Motivation 4
1.2 Problem Statement 7
1.3 Approach 9
1.4 Contribution 13
1.5 Structure of the Dissertation 13

2 Ubiquitous Computing Systems 15
2.1 Ubiquitous Computing 15
2.2 Socio-technical Computing Systems 16
2.3 Context-aware Computing 18
2.4 Adaptive Software Systems 19
2.4.1 Autonomic Computing and Self-Adaptive Systems 19
2.4.2 Adaptive User Interfaces 21
2.4.3 Socio-Adaptive Systems 22
2.5 Human-Computer Interaction 22
2.5.1 Usability 23
2.5.2 User Interruption 23
2.5.3 Trust 25
2.5.4 Individualisation 28
2.6 Summary 31

3 Self-Adaptive Software 33
3.1 Definition and Goals 34
8.2.2 Planning .................................................. 90
8.2.3 Execution ............................................... 92

9 Notification and Feedback ............................ 95
9.1 Notifications ............................................. 95
9.2 Notification and Feedback Management ............ 97
9.3 Summary ................................................. 100

10 Temporal User Participation ......................... 103
10.1 Pre-Adaptation Controllability ....................... 103
10.2 Post-Adaptation Controllability ..................... 104
10.3 Manual Adaptation ..................................... 105
10.4 Starting and Stopping Adaptation .................... 107
10.5 User Focus Adaptation .................................. 110
10.6 Detecting the User's Activity ....................... 114
10.7 Summary ................................................. 119

11 Behavioural User Participation ....................... 121
11.1 Dynamic Utility Functions ............................ 121
11.1.1 Types of Utility Functions ....................... 123
11.1.2 Building Dynamic Utility Functions ............ 124
11.1.3 Evaluating Dynamic Utility Functions .......... 127
11.1.4 Switching Dynamic and Static Utility Functions 129
11.1.5 Summary ............................................. 130
11.2 Fuzzy-Logic-Based Adaptation Reasoning .......... 130
11.2.1 Fuzzy Profiles ..................................... 132
11.2.2 Fuzzification ....................................... 133
11.2.3 Inference and Defuzzification .................... 135
11.2.4 Summary ............................................. 136
11.3 Learning-Based Adaptation Reasoning .............. 136
11.3.1 Learning Method .................................... 137
11.3.2 System Overview ................................... 138
11.3.3 Specifying Situations ............................... 139
11.3.4 Identifying Application Variants ............... 141
11.3.5 Sampling and Training ............................ 142
11.3.6 Classifying Situations ............................. 143
11.3.7 Example ............................................. 145
11.3.8 Summary ............................................. 147

III Evaluation ................................................. 149

12 Case Study: Meet-U .................................... 151
12.1 Meet-U .................................................. 151
12.2 Initial Study ........................................... 154
12.3 Evaluation Study ...................................... 156
12.3.1 Method and Realisation ........................... 157
12.3.2 Results .............................................. 161
### List of Figures

1.1 The MAPE-K adaptation loop integrated in an autonomic manager. . . . 3
1.2 Concepts for user participation in three dimensions. . . . . . . . . . . . . 10
1.3 Integration of user participation into the MAPE-K adaptation loop. . . 11

3.1 High-level view on self-adaptive software. . . . . . . . . . . . . . . . . . 33
3.2 Taxonomy of the self-adaptation domain. . . . . . . . . . . . . . . . . . 35
3.3 The three basic types of control loops in a control system. . . . . . . . 39
3.4 Feedback loops with user participation. . . . . . . . . . . . . . . . . . . 40
3.5 Internal vs. external adaptation approach. . . . . . . . . . . . . . . . . . 41
3.6 The MAPE-K adaptation loop as proposed for autonomic computing. . 43
3.7 State-Action model for adaptation planning. . . . . . . . . . . . . . . . . 44
3.8 Modelling variability. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

4.1 Analysis of research and application domains. . . . . . . . . . . . . . . . 58
4.2 Maturity of provided examples. . . . . . . . . . . . . . . . . . . . . . . . 59
4.3 Expected impact on users. . . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.4 Assessed quality of evaluations. . . . . . . . . . . . . . . . . . . . . . . . 60

6.1 Fuzzy control system. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
6.2 Common types of membership functions of a fuzzy variable $x$. . . . . . 71
6.3 AND, OR, and NOT operations on linguistic terms. . . . . . . . . . . . . 73
6.4 Exemplary fuzzification of the fuzzy variables $\text{temp}$, $\text{time}$, and $\text{heating}$. . 74
6.5 Accumulation and defuzzification result using the COG method. . . . . . 75

7.1 Integration of user participation into the MAPE-K adaptation loop. . . 79
7.2 Dimensions of user participation. . . . . . . . . . . . . . . . . . . . . . . 80

8.1 MUSIC system architecture. . . . . . . . . . . . . . . . . . . . . . . . . . 85
8.2 Types, components, and plans in MUSIC. . . . . . . . . . . . . . . . . . 87
8.3 High-level perspective MAPE-K self-adaptation in MUSIC. . . . . . . . 89

9.1 Schematic representation of different notification types with controllability. 96
9.2 Integration of the notification and feedback managers into the middleware. 98

10.1 Three application variants and their default degree. . . . . . . . . . . . . 109
10.2 Interplay between the user and the adaptive system. . . . . . . . . . . . . 111
10.3 Non-optimal decision with brute force reasoning and user focus plans. . 113
10.4 Determining a user’s interaction activity. . . . . . . . . . . . . . . . . . 116

11.1 Basic types of utility functions. . . . . . . . . . . . . . . . . . . . . . . . 125
11.2 Middleware user interfaces for managing utility functions on Android. . 127
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.3 User-oriented fuzzy-logic-based adaptation reasoning.</td>
<td>132</td>
</tr>
<tr>
<td>11.4 Middleware user interface for managing fuzzy profiles.</td>
<td>133</td>
</tr>
<tr>
<td>11.5 Fuzzification of context dependencies and fuzzy metrics.</td>
<td>134</td>
</tr>
<tr>
<td>11.6 Two-stage process for learning application variants.</td>
<td>139</td>
</tr>
<tr>
<td>11.7 Middleware user interfaces for managing situations on Android.</td>
<td>140</td>
</tr>
<tr>
<td>11.8 Parabolic function to weigh a situation value.</td>
<td>144</td>
</tr>
<tr>
<td>12.1 The Meet-U application.</td>
<td>151</td>
</tr>
<tr>
<td>12.2 Examples for different types of variability in Meet-U.</td>
<td>153</td>
</tr>
<tr>
<td>12.3 Five different design variations for the outdoor navigation.</td>
<td>159</td>
</tr>
<tr>
<td>12.4 The remote control application used by the supervisor.</td>
<td>160</td>
</tr>
<tr>
<td>12.5 Which design variations did participants prefer for which adaptation?</td>
<td>162</td>
</tr>
<tr>
<td>12.6 Would participants use an application that works only autonomously?</td>
<td>163</td>
</tr>
<tr>
<td>12.7 Which information would participants provide to improve adaptation?</td>
<td>163</td>
</tr>
<tr>
<td>12.8 Do participants want to individualise the application behaviour?</td>
<td>164</td>
</tr>
<tr>
<td>12.9 What type of interface do participants prefer for individualisation?</td>
<td>164</td>
</tr>
<tr>
<td>13.1 Demonstrator A: User interface realisations with notification and feedback</td>
<td>168</td>
</tr>
<tr>
<td>13.2 Demonstrator B: User interface realisations marked as user focuses.</td>
<td>170</td>
</tr>
<tr>
<td>13.3 Demonstrator C: Using the interaction activity level for adaptation.</td>
<td>172</td>
</tr>
<tr>
<td>13.4 Demonstrator C: Implementation</td>
<td>173</td>
</tr>
<tr>
<td>13.5 Demonstrator D: Specification of quality metrics.</td>
<td>174</td>
</tr>
<tr>
<td>13.6 Demonstrator D: Specification of multiple static utility functions.</td>
<td>176</td>
</tr>
<tr>
<td>13.7 Demonstrator E: Specification of the utility function with fuzzy metrics.</td>
<td>176</td>
</tr>
<tr>
<td>13.8 Demonstrator F: Specification of default components.</td>
<td>178</td>
</tr>
<tr>
<td>13.9 Demonstrator F: Sampling training data by preferring individual plans.</td>
<td>178</td>
</tr>
<tr>
<td>C.1 Taxonomy of the MUSIC adaptation approach.</td>
<td>191</td>
</tr>
<tr>
<td>C.2 The extended MUSIC UML modelling profile.</td>
<td>192</td>
</tr>
<tr>
<td>D.1 Two composite realisations of the Meet-U application.</td>
<td>193</td>
</tr>
<tr>
<td>D.2 Component type realisation of the Meet-U application.</td>
<td>194</td>
</tr>
<tr>
<td>D.3 Parameter settings for parametric adaptation in the Meet-U application.</td>
<td>194</td>
</tr>
<tr>
<td>D.4 Variability model for the <strong>HelloWorldApp</strong> application.</td>
<td>195</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Degrees of automation. . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
6.1 Linguistic term operations available in the FCL. . . . . . . . . . . . 73
6.2 Accumulation methods supported by the FCL. . . . . . . . . . . . . 76
6.3 Defuzzification methods supported by the FCL. . . . . . . . . . . . . 76
9.1 Reasons for notification. . . . . . . . . . . . . . . . . . . . . . . . . . . 99
9.2 Types of feedback. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
10.1 Interaction activity classes. . . . . . . . . . . . . . . . . . . . . . . . . . 115
11.1 Specification of quality metrics. . . . . . . . . . . . . . . . . . . . . . . 126
11.2 Example of a knowledge base after short training. . . . . . . . . . . . 145
11.3 Classification based on the observed data using the k-NN algorithm. . 145
12.1 Assignment of groups of participant to design variations and adaptations. 158
13.1 The different demonstrators and the mechanisms for user participation. . 167
13.2 Required bundles for the extended MUSIC middleware. . . . . . . . . . 179
B.1 Application examples provided by publications on self-adaptation. . . . 189
List of Listings

6.1 Simple rule base used by a fuzzy controller to adjust the heating in a room. 72

8.1 Example for addressing microphone context information with the semantic concepts Entity, Scope, Value Scope, and Representation. 90

8.2 Utility function for the MUSIC pilot application Travel Assistant. 91

13.1 Demonstrator A: The PortraitUI implementation using the notification and feedback mechanisms as well as simulated method calls for sending feedback and handling notifications. 169

13.2 Demonstrator B: User focus notification handling in the PortraitUI implementation. 171

13.3 Demonstrator D: Two static utility functions with UtilityFunction1 being the default function. 175

13.4 Demonstrator E: A utility function using fuzzy metrics and plan properties. 177

A.1 FCL function block for the heating example in Chapter 6. 187
List of Algorithms

8.1 The brute force adaptation reasoning algorithm. . . . . . . . . . . . . . . 91

10.1 Brute force reasoning with user-preferred plans (manual adaptation). . . 106
10.2 Calculation of the default degree of a configuration template hierarchy. . 109
10.3 Brute force reasoning in combination with default template reasoning. . 110
10.4 The user focus adaptation reasoning algorithm. . . . . . . . . . . . . . . 114

11.1 Run-time evaluation of dynamic utility functions. . . . . . . . . . . . . . . 128
11.2 Adaptation reasoning based on the k-nearest neighbour classification. . 146
List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Autonomic Computing</td>
</tr>
<tr>
<td>AOP</td>
<td>Aspect-oriented Programming</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASPL</td>
<td>Autonomic Software Product Lines</td>
</tr>
<tr>
<td>AUI</td>
<td>Adaptive User Interfaces</td>
</tr>
<tr>
<td>DSPL</td>
<td>Dynamic Software Product Lines</td>
</tr>
<tr>
<td>DUF</td>
<td>Dynamic Utility Function</td>
</tr>
<tr>
<td>FCL</td>
<td>Fuzzy Control Language</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HCI</td>
<td>Human-Computer Interaction</td>
</tr>
<tr>
<td>k-NN</td>
<td>k-nearest neighbour</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-driven Architecture</td>
</tr>
<tr>
<td>MDD</td>
<td>Model-driven Development</td>
</tr>
<tr>
<td>MUSIC</td>
<td>Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>QM</td>
<td>Quality Metric</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RFID</td>
<td>Radio Frequency Identification</td>
</tr>
<tr>
<td>RL</td>
<td>Reinforcement Learning</td>
</tr>
<tr>
<td>SAS</td>
<td>Self-adaptive Software</td>
</tr>
<tr>
<td>SEAMS</td>
<td>Software Engineering for Adaptive and Self-Managing Systems</td>
</tr>
<tr>
<td>SOA</td>
<td>Service-oriented Architecture</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>SOC</td>
<td>Service-Oriented Computing</td>
</tr>
<tr>
<td>SPL</td>
<td>Software Product Lines</td>
</tr>
<tr>
<td>SUF</td>
<td>Sub-Utility Function</td>
</tr>
<tr>
<td>UAD</td>
<td>User (Interaction) Activity Detection</td>
</tr>
<tr>
<td>UC</td>
<td>Ubiquitous Computing</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>ULSS</td>
<td>Ultra-Large-Scale Systems</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>VENUS</td>
<td>Gestaltung technisch-sozialer Vernetzung in situativen ubiquitären Systemen</td>
</tr>
</tbody>
</table>
Abstract

Self-adaptive software provides a profound solution for adapting applications to changing contexts in dynamic and heterogeneous environments. Having emerged from Autonomic Computing, it incorporates fully autonomous decision making based on predefined structural and behavioural models. The most common approach for architectural run-time adaptation is the MAPE-K adaptation loop implementing an external adaptation manager without manual user control. However, it has turned out that adaptation behaviour lacks acceptance if it does not correspond to a user’s expectations – particularly for Ubiquitous Computing scenarios with user interaction. Adaptations can be irritating and distracting if they are not appropriate for a certain situation. In general, uncertainty during development and at run-time causes problems with users being outside the adaptation loop. In a literature study, we analyse publications about self-adaptive software research. The results show a discrepancy between the motivated application domains, the maturity of examples, and the quality of evaluations on the one hand and the provided solutions on the other hand. Only few publications analysed the impact of their work on the user, but many employ user-oriented examples for motivation and demonstration. To incorporate the user within the adaptation loop and to deal with uncertainty, our proposed solutions enable user participation for interactive self-adaptive software while at the same time maintaining the benefits of intelligent autonomous behaviour. We define three dimensions of user participation, namely temporal, behavioural, and structural user participation. This dissertation contributes solutions for user participation in the temporal and behavioural dimension. The temporal dimension addresses the moment of adaptation which is classically determined by the self-adaptive system. We provide mechanisms allowing users to influence or to define the moment of adaptation. With our solution, users can have full control over the moment of adaptation or the self-adaptive software considers the user’s situation more appropriately. The behavioural dimension addresses the actual adaptation logic and the resulting run-time behaviour. Application behaviour is established during development and does not necessarily match the run-time expectations. Our contributions are three distinct solutions which allow users to make changes to the application’s run-time behaviour: dynamic utility functions, fuzzy-based reasoning, and learning-based reasoning. The foundation of our work is a notification and feedback solution that improves intelligibility and controllability of self-adaptive applications by implementing a bi-directional communication between self-adaptive software and the user. The different mechanisms from the temporal and behavioural participation dimension require the notification and feedback solution to inform users on adaptation actions and to provide a mechanism to influence adaptations. Case studies show the feasibility of the developed solutions. Moreover, an extensive user study with 62 participants was conducted to evaluate the impact of notifications before and after adaptations. Although the study revealed that there is no preference for a particular notification design, participants clearly appreciated intelligibility and controllability over autonomous adaptations.
Acknowledgements

When I began working on this dissertation in 2009, there was no road to take, there was not even a visible trail. Since then, a lot has changed and little by little a path appeared or it was built piece by piece. This work would not have been possible without the help of many people who contributed and supported over the last five years.

First and foremost, I want to thank my fiancée Sarah for her continuous support from the very beginning and in the end especially. Without her bothering constantly about the progress of my work, I probably would have lost track early. She was able to bring up the motivation in me that was required to finish this work.

Then of course, I would like to thank my supervisor Kurt Geihs for providing the opportunity of working in his department. With his outstanding expertise on self-adaptive software he provided important support and guidance throughout the process of the dissertation. Furthermore, I am very glad to have Albrecht Schmidt as my second adviser. I became acquainted with his excellent work on Ubiquitous and Pervasive Computing during my years of study in Essen. And of course, I am grateful to Ludger Schmidt and Gerd Stumme for completing the examination committee.

At the Distributed Systems Group at the University of Kassel, I was able to work with a lot of excellent people. Here I want to thank Stefan Niemczyk and Andreas Witsch in particular. All the fruitful discussions, the professional input, and the motivation helped a lot in finding my way. However, I do not want to forget to mention the rest of my (former) colleagues: Harun Baraki, Heidemarie Bleckwenn, Steffen Bleul, Diana Comes, Tareq Razaul Haque, Mohammad Ullah Khan, Dominik Kirchner, Thomas Kleppe, Stephan Opfer, Roland Reichle, Iris Roßbach, Daniel Saur, Hendrik Skubch, Michael Wagner, Thomas Weise, and Michael Zapf. Not to forget the student assistants from which I want to mention Jonas Scherbaum, Philipp Faßheber, and Philipp Sperber especially.

Between 2010 and 2013, I had the opportunity to work within the multidisciplinary research cluster VENUS. It was a great experience working in such a diversified research group with members from different fields of research. Together with Romy Kniewel from the Human-Machine Systems Engineering Group, a very fruitful collaboration was founded. Several joint publications attest the success of this collaboration. Thank you very much for the discussions and cooperation within VENUS and the Meet-U user study.

Last but not least, a special thank you to all the people who were proofreading this dissertation: Philipp Dörr, Armin Evers, Ulrich Evers, Stefan Niemczyk, Daniel Saur, Michael Wagner, and Andreas Witsch. Every piece of input and feedback was very helpful. And not to forget I would like to thank my parents and grandparents for having their support whenever it was necessary.
Part I

Foundations
Socio-technical computing systems provide intelligent application behaviour and anticipation to users’ needs while simultaneously considering social constraints of the target group or legal implications. This applies particularly to ubiquitous computing systems which are significantly intertwined with people’s everyday life. Since Weiser [159] proposed the concept of ubiquitous computing and the associated ideas in the early 1990s, researchers became inspired and developed concepts to realise his vision. For example, self-adaptive software addresses intelligent automation and anticipation.

Unlike traditional software, self-adaptive software can change its state and behaviour at run-time. Application-level adaptation is a result of autonomic computing research [106] to achieve the best desirable service for a user without involving the user in the actual adaptation decision. The MAPE-K adaptation loop [75] is the most popular attempt to incorporate the self-adaptation concept in software for automation of complex software infrastructures in dynamic environments (cf. Figure 1.1). However, autonomous application behaviour may have serious impact on users and it does not always meet the user’s expectations.

![Autonomic Manager](image)

**Figure 1.1**: The MAPE-K adaptation loop integrated in an autonomic manager [106].

Let us consider a mobile application that supports the user navigating to a particular location. It chooses the routing algorithm based on the transportation means, e.g. whether the user is walking or using public transport. The self-adaptive application uses its sensors and user history data to determine which transportation means is actually used and selects an appropriate application configuration. In case the transportation means was chosen wrongly, the user will not reach the location in time. It further may be distracting if the application configuration changes during navigation, for example when the self-adaptation algorithm changes map or routing information without prior notice.

The automation paradox, also called *irony of automation* [11], is known since automated control systems took over tasks that have been previously carried out by human operators. Psychologists identified human contribution in automated systems not less but more important. The more advanced the automated systems is the more crucial the contribution of the human operator will be. In case of failures or irregular conditions
humans still have to intervene. Later on, researchers tried to re-integrate human users into the decision loop to let operators acquire regular experience with the system and achieve a more efficient overall system performance [38]. By now, there has been only little effort to consider human factors in self-adaptive software.

We will discover that on the one side there is this thorough and comprehensive research on human factors in automated computing systems and on the other side there is this field of self-adaptive software – emerged from automated systems – often disregards the user; although continuously claiming to provide the best service and satisfaction to the user. Considering the argumentation for self-adaptive and autonomic computing is in favour of automation with the user removed from the loop as much as possible, we will reveal that the majority of research addresses application domains that inherently require users within the loop. We do not question self-adaptation and autonomic computing in general, but rather propose an alternative concept with an opened adaptation loop for user participation in applications with user interaction.

1.1 Motivation

Smart homes, smart mobile applications, intelligent vehicle systems, or intelligent wearable devices are prominent examples for ubiquitous computing applications that all include intelligent or smart in their name. Context awareness and autonomic computing provide an essential basis for intelligent systems that are able to adjust themselves to changing run-time conditions. However, in all these application domains the user takes an important role. Sometimes the user is only consuming information while more often he actively interacts with the system.

Another motivation for self-adaptive software is an increase in complexity that is hard to cope with. Automated and simplified management is required [186]. However, others argue that autonomic computing makes systems complex and opaque to users [6]. So ironically, autonomic computing may provoke the problem which it tries to solve; at least in user-oriented domains like ubiquitous computing. Van der Heijden [56] argues that transferring control from the user to the system results in increased user anxiety. Barkhuus and Dey [14] revealed in a study on context-aware applications that users preferred context-aware features over personalisation, but in the same moment they experienced a loss of control. In adaptive user interface research, Weld et al. [160] warn that adaptivity (for better personalisation) has the potential to create confusing inconsistency and rob the user of control.

Decreasing controllability of systems can decrease users’ trust. According to Söllner et al. [139], users may further not trust such applications and conceal their actions. Russell et al. [127] refer to ghosts in the machine when speaking about autonomous systems: “when systems manage their own very high-level tasks, their behaviors and the interaction of their behaviors might be unpredictable and inscrutable, complicated, and hidden from view. From a user's perspective, an autonomous system might seem to have ‘magical’ properties, where things happen without apparent or determinate causes”. With self-adaptation “the system no longer exhibits a direct cause-and-effect relationship between commands and actions” [73]. Due to autonomous reasoning on environmental changes it may not be clear to the user why something happens when it happens.
Often software designers develop autonomous systems to explicitly exclude the user. They know that people make mistakes and try to design systems that reduce the need for human intervention. However, it is hard to automate complex tasks and many times only simple, easy to automate tasks are adopted by the autonomous system. The difficult and error-prone tasks stay with users or operators of the system. Due to the lack of constant hands-on experience with simple tasks, the human is taken out of the loop and such systems become harder to use. Automation can actually decrease system transparency and increase system complexity. Users who got out of the loop are no longer attentive or aware of the current operating context [127]. Also, Salehie and Tahvildari [128] state that human involvement in general is quite valuable for improving the manageability and trustworthiness of self-adaptive software. Others stress the importance of cooperation between human users and operators and the adaptive system [40, 48]. Instead of taking the user out of the loop, it may be more efficient for the system's overall goal to use synergy effects between human and machines. Anderson et al. [6] are even more drastic: “no system, therefore, can be wholly autonomic as there will be at some stage the need to have a human user input to decide [ . . .]”. Regarding autonomous policy reasoning they state that “[ . . .] there are exceptions [to autonomous reasoning] and humans are best at coping with these”.

Automation and intelligent adaptation behaviour are an important aspect of ubiquitous computing. However, ubiquitous computing aims at human-computer interaction, too. With computing systems that involve significant amounts of user interaction, the user needs to be integrated in the decision-making process.

When humans use computing systems they have expectations on the system's functionality and behaviour. In traditional software engineering processes such user requirements and constraints are determined during a requirements analysis phase (in addition to requirements from all other stakeholders). As the actual software system is static and predictable it cannot be expected to behave differently unless the software incorporates a faulty behaviour or other mistakes. On the contrary, adaptive software systems act autonomously during run-time. They adapt their structure, behaviour, or functionality to changing run-time conditions depending on the implemented decision algorithms and knowledge base. The actual behaviour cannot be completely foreseen at design-time and it is difficult for the application developer to anticipate all possible outcomes and situations that may occur during execution. Moreover, the degrees of freedom may be user-specific and not target-group-specific as in traditional software. Different users have different expectation on the software, especially when bearing in mind the individual and personal character of typical ubiquitous computing scenarios. Applications often work on personal preferences in the individual environment of the user. Even for a single user the user's behaviour is not predictable in advance; users start new routines or new life situations [125]. Such changes may alter the user's view on the software system. In static software systems individual differences between users are adjusted with configuration settings. One can say that we just need a better requirements engineering process to match the more complex requirements to the software. However, it is already cumbersome to deal with complex adaptation architectures without respecting changing user requirements. Further, preferences for adaptation mechanisms are different to preferences in static software as we have to interfere with an autonomous decision engine. Autonomous adaptation processes can have serious impacts on the system's usability. Executing user-interface-related adaptations has a direct effect on the user
experience, either positive or negative. Adaptations can occur when users are involved with their current tasks. In order to inform the users on the on-going adaptation, their attention has to be drawn from those tasks to the particular adaptation. As attention is a limited resource of human-computer interaction [63], the costs and benefits of drawing the user’s attention have to be balanced. Otherwise the application would not meet the user’s goals, and thus become unusable. From a user’s perspective, adaptations can be distracting, intrusive, and interrupting [97]. Typically, interruptions are notification messages which can be a problem when they do not support the user’s current activity [122] [125].

Beside usability issues and social factors like trust, transparency, or privacy, self-adaptive systems bear several self-made issues founded in the technological concept. The idea behind self-adaptive software is the ability to autonomously perform changes in parameterisation or configuration in reaction to contextual input and associated rules or other decision techniques. Although not every single change in the context of a user may be relevant to an application, some situations like the presence of certain people, services, environmental conditions like location or temperature, or derived information about people’s current activity and appointments from calendars may affect the application’s behaviour. However, this process may include several sources for uncertainty. First, the contextual input may be erroneous or inaccurate and therefore the adaptation reasoning could come to improper results that are not suitable for the current situation. And second, context sensors might not represent the user’s reality correctly or it is very hard to precisely define the necessary contextual information. The reasoning process may add additional uncertainty.

Once a self-adaptive application is executed, user interaction with the system is limited. The possibilities for interaction depend on the decisions made by the system. Users typically cannot switch to a different application variant, undo adaptations, or postpone adaptations. Controlling or influencing the self-adaptive decision loop is not possible. User participation and human-computer interaction (HCI) is seen as a challenge for adaptive software system research ever since; Cheng et al. [27] listed human-computer interaction as one of their main challenges for the roadmap in self-adaptive applications. They recommend integrating the user in the adaptation loop, which has not been adequately addressed so far.

Salehie and Tahvildari [128] define human-system interaction as one of the main challenges in self-adaptive software research: “[such] challenges relate to building trust, providing an appropriate mechanism for collecting user policies, and establishing a proper mechanism to involve humans in the adaptation loop”. The authors further summarise that “[…] most of the projects do not have a human in the loop for policy changing or tracing adaptation processes”.

Weyns et al. [163] did a systematic study on the claims associated with self-adaptation and the evidence that exists for these claims. They found out that most of the research is on architecture and models and that its authors do not consider limitations and impact of their work. The primary considered software quality attributes are flexibility, reliability, and performance. The tradeoffs implied by self-adaptation have not received much attention. Many of the example applications provided in self-adaptive research require user participation and interaction, but it was not considered during concept and application development. In an extended study with the focus on architecture-based self-
adaptation [162], Weyns and Ahmad further found that most of the examples are simple example applications with a minimal level of evidence. Real industrial applications are rarely provided. Latest research for self-adaptive systems claims the improvement of single software qualities like performance, reliability, or flexibility. Tradeoffs like usability, human-computer interaction, transparency, or user participation are hardly considered at all.

When we did a first usability analysis with our elaborated mobile Meet-U application [214] within the scope of the VENUS project, we quickly discovered serious usability issues in combination with an architectural adaptation approach [219]. In summary, the user seems to be out of the loop – whether it is during software development or run-time.

Originally, autonomic computing and self-adaptation are supposed to consider no or only little user participation and interaction. With all the challenges and problems mentioned, one might ask if user consideration in autonomic computing is a reasonable approach. However, the benefits of autonomic computing are evidentiary and researchers agree on the necessity for user participation.

1.2 Problem Statement

We claim that self-adaptation in ubiquitous computing systems works as long as the user is not negatively influenced and constrained by autonomous processes. Users appreciate autonomous processes if they actively support the user to achieve his current tasks. They are impressed if a computing device can foresee possible actions and act according to the forecast. However, they feel distracted and lose control if adaptive processes do not match their expectations. Hence, the overall objective and research question we would like to answer is:

**How can we enable user participation in interactive self-adaptive software while at the same time maintaining the benefits of intelligent autonomous behaviour?**

We base this work on preliminary work done by the MUSIC initiative [187]. Thus, we focus on component-based architectural adaptation [189, 199] applied to mobile applications in heterogeneous ubiquitous computing environments. The considered applications typically have a high degree of user interaction. MUSIC provides a model-driven development methodology and a self-adaptation middleware that separates self-adaptation capabilities from actual application logic. We define the following problems for this kind of computing system:

**Main Problem 1 – Uncertainty:** The execution environment is uncertain and dynamic. It cannot be properly modelled at design time of the system to perfectly fit user requirements. This has several reasons:

1. **Static decision making:** Even if the application seems to operate dynamically at run-time, its decisions are based on an (structural) adaptation model with associated behavioural information (e.g. rules or a utility function). These specifications are static at run-time and cannot be changed, e.g. by the user. Binding external services introduces more flexibility, but unanticipated integration of services is still very challenging.
2. **Sensor data and reasoning:** Decisions are based on context data and such data is not necessarily precise. Sensors can deliver erroneous data without noticing by the system or they even fail. Reasoners may have problems with anomalous input and may calculate wrong values.

3. **Imprecise analytical models:** Context sensors and reasoners may not properly grasp the user's context as the underlying analytical model does not fit. Even though sensors deliver perfectly correct data, this data may not represent the user's reality correctly. For example, reasoners might not be specified for a particular input range.

4. **Uncertain human behaviour:** Even if the decision engine provides precise results, it cannot cope with the uncertain human nature. If it comes to application-level or user-interface-related adaptation, a user might not always prefer the same behaviour under all circumstances. Human decision-making is based on many parameters (the social context particularly) that cannot be easily represented by an adaptation model.

5. **Device heterogeneity:** Devices for end-user access may vary in software and hardware configuration. They may differ in screen display, available input methods, or supported sensors. Especially sensors provide different contextual information with different scales or representations. Some information will not even be available on some platforms. Adaptive software, developed for multiple device types might not be able to grasp the user's contextual situation properly as context reasoners may depend on some very specific sensor information.

Our aim is not to make sensors more reliable or human behaviour less uncertain. Further, it is virtually impossible for an application developer to cover all conceivable user situations in advance. We propose that a user should be able to modify and influence a running application to cope with uncertainty originated during development or run-time.

**Main Problem 2 – Controllability:** Self-adaptation implies no explicit and implicit influence or control by humans during execution. Once executed, self-adaptive software may not allow any further changes. A user has to rely completely on the competence of the development team. Even if people sometimes like context-aware and adaptive features, they become anxious due to the loss of control.

**Main Problem 3 – Trust:** From the loss of control may follow lack of trust in self-adaptive software. Autonomous systems that are used, observed, or administered by humans are hardly intelligible and users do not trust when they do not know why or why not adaptations occur. Further, users do not know when things will happen and why they happened at a certain moment. Consequently, autonomous actions may decrease trust in such systems.

**Main Problem 4 – Usability:** Adaptation raises serious usability issues. Users become distracted from their current tasks when adaptations occur without prior notification. Even with notifications, a user can get bothered by adaptation attempts. As long as adaptations only adjust backend components or quality of service parameters, the user is only affected indirectly. However, adaptations that involve the user interface of an application have to be developed thoroughly. In the worst case, the autonomous system decides to adapt the user interface while the user is interacting with it in the same moment, e.g. while reading or writing from/to the user interface.
Main Problem 5 – Adaptive systems for multiple users: In times of tablet computers, people share their devices among friends and family. But also other ubiquitous computing applications like smart homes involve multiple users. Context-aware and self-adaptive software can be very specific to individual user preferences. People use the same applications, but have entirely different requirements to an application. An adaptive application needs to be adaptive to different users, either by being adjusted or by learning the user’s requirements.

To summarise the above problems we can say that it is necessary to let the user take part in the adaptation loop of autonomous systems and to consider his interaction habits as well as his personal preferences for adaptation reasoning. Further, usability and interaction design standards have to be respected to increase trust, controllability, and general acceptance of such systems.

1.3 Approach

To address the five main problems from a software engineering perspective we provide mechanisms for participation and modification by the user once the application has been deployed and executed. We assume that explicit user participation countervails uncertainty, increases trust and controllability, and allows for multiple users in adaptive systems. With the provision of bi-directional communication between the adaptive system and the user as well as the consideration of the user’s interaction habits, we will further improve the usability of such systems.

We first identify all options and parameters that would allow us to influence an adaptive system from the system’s perspective. Independent of the concrete implementation, we define three dimensions of user participation: behavioural, temporal, and structural user participation (cf. Figure 1.2). Each dimension has a scale starting with “the system decides” and ending with “the user decides”. Our aim is to adopt a compromise between the two extremes.

1. **Temporal**: The temporal dimension determines the entity which controls the moment of adaptation, i.e. which entity triggers the adaptation process. In a MAPE-K adaptation loop, the system analyses information retrieved from sensors or reasoners and the planning engine decides which variant of the software will be selected for instantiation. The user might not always be comfortable with a certain moment of adaptation and its consequences when the system has made the decision on its own.

2. **Behavioural**: The behavioural dimension describes the adaptation logic. At design-time a developer typically specifies the system’s behaviour based on an adaptation policy according to the available components, parameters, and context values. This approach can be too static for some applications, especially when the user wants to control the application behaviour according to his preferences and needs. It may be also necessary to have influence on the decision subject when the developer-defined behaviour is not sufficient.

3. **Structural**: The structural dimension addresses the components, modules, and services of an application and the entities that provide them. Components or
services can be either supplied by the software developer or provider of the application. They can be further dynamically integrated from third-party providers via standardised interfaces at run-time. In both cases the question is whether the user or the system decides which of the components or services are actually used.

<table>
<thead>
<tr>
<th>Temporal</th>
<th>Behavioural</th>
<th>Structural</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual selection of</td>
<td>Modification of utility functions</td>
<td>Selection of components available for adaptation</td>
</tr>
<tr>
<td>application variants</td>
<td>Support for behaviour profiles</td>
<td>Selection of external services</td>
</tr>
<tr>
<td>Controllability of</td>
<td>Simpler adaptation reasoning specification</td>
<td>Modification of the architectural composition</td>
</tr>
<tr>
<td>autonomous adaptations</td>
<td>Learning of individual user behaviour</td>
<td>Support for end-user development</td>
</tr>
<tr>
<td>Toggle self-adaptation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mechanism</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consider the user’s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>current focus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consider the user’s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interaction activity</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Figure 1.2: Concepts for user participation for the behavioural, temporal, and structural dimensions. Notification and feedback provide the basis for all dimensions.](image)

In this work we provide solution concepts for the behavioural and temporal dimensions only and leave the structural dimension for future work. From a user’s perspective, we distinguish between short-term and long-term participation. By short-term we mean a situation in which the predefined application behaviour is modified by the user by simply interacting normally with it. A user alters the adaptation behaviour ad-hoc and only for this very moment. In such a case the user is also not aware of actually modifying the behaviour. Short-term participation is a deviation from the system’s implemented default behaviour. Long-term participation characterises adaptation control when the user is aware of modifying the application’s behaviour, e.g. by altering preferences or by specifying new adaptation rules. These changes are persistent and alter the system’s implemented default behaviour in the long run.

User participation requires intercepting the adaptation control loop which is the basis for autonomous decision-making in self-adaptive software. User participation further requires bi-directional communication between the user and the adaptive system. In this work we extend the MAPE-K adaptation loop with capabilities for user participation (cf. Figure 1.3). This general approach allows for the implementation of user participation in different systems based on the MAPE-K loop. User participation is only relevant for the planning component of the loop. Within this component the actual adaptation reasoning happens, i.e. the adaptation reasoner plans which application variants will be instantiated in the subsequent execution component. A notification component allows the reasoner to communicate with the user, for example when it anticipates an adaptation to be
unexpected; then the user can react accordingly. For this type of short-term modification, a feedback component mediates between the user and the adaptation reasoner. Feedback is not dependent on prior notifications. A user can change application variants, undo and postpone adaptations, or toggle overall adaptation at any time.

![Participation Management Diagram](image)

**Figure 1.3:** Integration of user participation into the MAPE-K adaptation loop.

The temporal moment of adaptation has a significant influence on the usability, the perceived trust, and controllability of the system. Hence, we integrate two mechanisms to improve those factors. First, we define the user focus attribute of component realisation plans. This attribute indicates if a component needs a special consideration during reasoning, for example to avoid distraction of the user from his current task. Second, we include the user's current activity in the reasoning process, e.g. to avoid interface adaptations when the user is actively interacting with the system.

For long-term modification, a different channel exists: a user has to modify the decision and knowledge base from which the planning engines take its standard behaviour. Depending on the implemented reasoning approach, the user modifies utility functions, rules, or knowledge bases for learning algorithms.

The MUSIC approach adopts utility-function-based adaptation reasoning. Utility functions are known to be hard to create and maintain by experts when there is no user participation and interaction yet [184, 43, 72]. Shifting this task to the actual non-expert user is not a promising concept. However, we exploited the possibility of modifying utility functions at run-time for developers and expert users by providing abstractions to the mathematical concept. The basic idea is to reduce mathematical complexity by splitting a single utility function into multiple sub-functions, one for each quality metric. All sub-utility functions will be combined to a single overall function. Further, we provide predefined types of functions, e.g. linear or sigmoidal functions. To avoid constant modification of utility functions, users can save utility functions as behaviour profiles to cope with changing run-time situations. Moreover, those utility functions can be dynamically switched by the user while the software is running. The new behaviour will be adopted instantly.

As utility functions are difficult to grasp for ordinary users, we developed a more user-friendly approach based on Fuzzy Logic [170]. Fuzzy logic allows robust reasoning on diffuse, i.e. fuzzy values based on Fuzzy Sets [171]. Such values are evaluated by rules.

1.3 Approach  11
that can be easily maintained and created by non-experts. Instead of building rules with traditional logic expressions, fuzzy logic uses natural language terms. We have integrated fuzzy reasoning into the MUSIC middleware and provide a user interface for fuzzy profile management and rule modification by users.

A third solution for long-term participation is based on machine learning techniques with a k-nearest neighbour classification algorithm. Developers and users can define situations depending on context information provided by the context middleware. Here we employ the same fuzzy reasoning approach as described before to specify which context parameters define a situation. During training mode, the user has to select application variants manually, i.e. there is no autonomous operation at this time. The system stores the current situation vector for this application variant. In normal operation mode, the current situation is compared to the stored application variant entries employing a k-nearest neighbour classification. We select the best fitting application variant for this particular situation according to the users’ learned behaviour. We make use of the feedback and notification mechanism during training and operation mode.

Although the presented solution assumes an underlying middleware- and component-based solution for architectural adaptation, the generic concept can be transferred to other adaptation domains. Further, we focus on user acceptance emerging through usability. We define the following assumptions and limitations:

1. The adaptive system follows a component-based architectural adaptation approach. Although requirements-driven approaches are promising, architectural adaptation still prevail [7].

2. The adaptive system makes use of the MAPE-K adaptation loop [75].

3. Applications have a significant amount of interaction. We do not consider systems that mainly operate autonomously with only little time of user interaction. Hence, we do not consider systems that only perform quality of service optimisations.

4. Achieving transparency and intelligibility in adaptive systems is important, but we do not want transparency under all circumstances. For example, we assume that making context information visible to users actually reduces intelligibility of the system.

5. The adaptive system is used by a single user at a time. Conflicting user interests are left for future work.

6. The goal of adaptation is not to improve usability alone, but rather to improve user experience and other non-functional requirements like performance, reliability, or efficiency at the same time.

7. Although privacy and security issues need to be considered for user acceptance in general, they are left for future work.

8. Interaction between the user and the adaptive system is important and interaction design and application design in general is necessary for increasing user acceptance. However, we omit the actual design part and refer to the work from Kniewel et al. [222] who focus on interaction design in adaptive applications.

9. We assume to generate trust in adaptive systems by user participation, however, we will not measure how much trust we have generated. We refer to Söllner et al. [139] for trust research in socio-technical systems.
10. This work does not deal with collaboration among human and software agents (e.g. autonomous robots) unless the human is using the robot as he would use an adaptive application.

### 1.4 Contribution

Our general contribution is the **systematic integration** of the user in self-adaptive applications by opening the closed MAPE-K adaptation loop and allowing bi-directional communication between the adaptive system and the user.

We identify **participation dimensions** of component-based applications which the user is able to influence and hence achieve a higher user acceptance.

Based on the dimensions of user participation we define several **mechanisms for active user participation** while still maintaining application autonomy. Such mechanisms address short-term and long-term modifications by users in the **temporal and behavioural dimensions**.

To **avoid user distraction, decreased trust, and loss of controllability**, we extend the decision capabilities of the adaptive system to respect the user’s perceptual focus and his activity while using the application.

Based on the MUSIC project, we provide a **self-adaptive middleware with support for user participation** nested in the extended MUSIC model-driven development methodology. The implementation follows a component-based architectural adaptation approach.

Based on the MUSIC methodology, we provide modelling and **model-driven development support** to develop adaptive applications with user participation.

The **Meet-U case study** demonstrates the feasibility of the concepts in a realistic mobile ubiquitous computing application. Six **demonstration applications** illustrate the practical implementation.

### 1.5 Structure of the Dissertation

The dissertation is divided into three parts. Part I describes the foundations and previous research which serve as the basis for the rest of the document (five chapters). Part II describes the concept and solution in detail (four chapters) and Part III evaluates and demonstrates the implemented concepts (three chapters). The content of the rest of the document is summarised as follows:

**Chapter 2** defines the overall setting in terms of ubiquitous computing systems. Furthermore, it provides required multidisciplinary research results from the field of human-computer interactions.

**Chapter 3** provides a general classification and an overview of the self-adaptation field. We analyse how well user participation can be achieved by different adaptation approaches, especially with different types of control loops.
Chapter 4 presents a literature study on frameworks, middlewares, and other systematic approaches to realise architectural self-adaptation. We reveal that most of the presented works use user-oriented examples to motivate their work and lack of proper evaluations.

Chapter 5 elaborates our study on related work.

Chapter 6 provides foundations on fuzzy control and fuzzy logic. We will use fuzzy control for our behavioural participation concepts.

Chapter 7 presents a general overview on our user participation concept and solution and how we define each of the user participation dimensions.

Chapter 8 embeds our solutions into the underlying MUSIC middleware and development methodology for architectural self-adaptation in mobile ubiquitous environments.

Chapter 9 introduces the basic notification and feedback concept that is used by the temporal and behavioural participation dimensions.

Chapter 10 provides our solution for the temporal user participation dimension.

Chapter 11 provides our solution for the behavioural user participation dimension.

Chapter 12 presents the Meet-U case study and an extensive evaluation of the notification and feedback concept with pre- and post-controllability.

Chapter 13 introduces six demonstration applications. Each demonstrates the feasibility of temporal and behavioural concepts and the notifications and feedback concept.

Chapter 14 finally concludes this dissertation and presents an outlook on future work.
2 Ubiquitous Computing Systems

The term *Ubiquitous Computing (UC)* has been introduced by Mark Weiser in 1991 [159]. He envisioned computing systems that move from the foreground into the environmental background of people’s focus. According to Weiser, people will shift their focus from single computing systems to many smaller and distributed computing facilities. This change in computing paradigm also includes a change in user interaction and communication among computing entities.

In this chapter we will introduce ubiquitous computing systems and related fields of research. UC systems cannot be precisely defined in a few sentences. Rather they need to be characterised with common properties that have evolved in the last two decades. Our focus is on context-aware and adaptive systems that establish the foundation for many UC application scenarios. However, UC systems are also socio-technical systems and to understand their full effect, we have to consider the social part, too. It is necessary to understand impacts from social sciences to the technical part of the system and vice versa. For the provision of user participation, we particularly include the human-computer interface and interaction design. The concerning disciplines have to be taken into account when analysing the effects adaptations have on the perceived application usability and trust.

2.1 Ubiquitous Computing

When Weiser wrote in *The Computer of the 21st century* [159] “The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it”, he probably did not expect the impact it would have on researchers worldwide. It was and still is the motivation for many related research projects.

From Weiser’s statement we can deduce that the provision and processing of information will be moved into the surrounding infrastructure. Information and services will be ubiquitously available. Technology is not anymore limited in spatial and temporal means. It is everywhere available and can be used by everyone. Customised services are offered and adapted to the needs of the individual user. A distributed network structure allows communication among many small entities and continuous exchange of information within the environment. Hence, ubiquitous computing dissolves “traditional boundaries for how, when, and where humans and computers interact” [98].

In ubiquitous computing, systems need to sense and adapt to the environment. They have to understand the context in which they are interested. This new class of computer systems, adaptive to the context, enables the development of more elaborated and complex applications. Such systems exploit the mobility of the user and the
dynamic nature of modern computing infrastructures. Nevertheless, the development of applications that continuously adapt to the environment and still running even when the user is moving or switching device, remains an open research challenge.

Due to the loosely definition of UC systems and related domains, we specify basic properties that such systems may provide:

- context awareness
- adaptivity
- distribution
- mobility
- service orientation
- environmental integration
- usability
- intelligence
- anticipation

In this work we focus on the software character of UC systems, i.e. we will provide guidance on how do we realise systems that fulfil the above criteria and how do we systematically develop such systems from a software engineering perspective.

Two other terms that are often used in the context of ubiquitous computing are Pervasive Computing and Ambient Intelligence. Those terms have been developed in different research communities, but basically mean the same and are used interchangeably nowadays\(^1\). Other closely related research areas are Wearable Computing, Internet of Things, Cloud Computing, and Nomadic Computing which we will not further discuss in this work.

While some research is done close to the original vision of UC (e.g. Smart Spaces [185, 142]), new fields came up to address the single properties of UC systems. In the following, we describe some of the properties in more detail as far as they are relevant for this work. We will especially focus on mobile, adaptive, and context-aware systems that require significant user interaction. Consequently, we will provide additional background information on research and techniques in the field of human-computer interaction research.

### 2.2 Socio-technical Computing Systems

Socio-technical systems (also: sociotechnical systems) have already been known before the technological predominance of computing systems in our everyday life, e.g. by Trist and Bamforth [149] who coined the term socio-technical system to describe the complex interaction structures between humans, machines, and the environment. While later on authors like Mumford [105] stress the organisational challenges socio-technical

\(^1\)Naturally, there are differences between Ubiquitous Computing, Pervasive Computing, and Ambient Intelligence depending on who one might ask. However, for this work we focus on the large set of common ground as described in this section.

16 Ubiquitous Computing Systems
computing systems impose, Baxter and Sommerville [15] focus on a holistic approach on socio-technical systems design. The latter pursues a balance between social/behavioural aspects and technical aspects of system development.

Badham et al. [10] define five key characteristics for socio-technical systems:

1. Systems should have interdependent parts.
2. Systems should adapt to and pursue goals in external environments.
3. Systems have an internal environment comprising separate but interdependent technical and social subsystems.
4. Systems have equifinality. In other words, systems goals can be achieved by more than one means. This implies that there are design choices to be made during system development.
5. System performance relies on the joint optimisation of the technical and social subsystems. Focusing on one of these systems to the exclusion of the other is likely to lead to degraded system performance and utility.

With the term *social subsystem*, Badham et al. refer to people, work context, and organisation.

UC systems impose a new proximity between computing systems and users and hence comply with the five key characteristics. UC systems are much more integrated into the social context of people, i.e. their everyday life. This brings new challenges regarding the systematic development of systems coping with this interplay of computing systems and people. For example, privacy and security become more important when systems and people are closely coupled. Moreover, user’s trust into UC systems is harder to achieve as systems make independent decisions and their actual behaviour is opaque. The dynamic structure of intelligent environments makes it hard for people to grasp the functionality and hence to build up trust easily. Further, the way people interact with UC systems has changed compared to classical desktop computing applications. Interaction designers and usability engineers need to find new best practices for human-computer interaction. If we further assume that all this happens with other people around, maybe using the same systems simultaneously, the social context has to become an important part of UC system design.

While classic software development approaches are not suitable for the development of ubiquitous computing systems, new approaches need to be developed to cope with the above challenges [49]. For the development of socio-technical systems it is essential to include stakeholders from a wide range of disciplines. However, disciplinary boundaries hinder the success of joint development teams [15].

Recently, researchers started to overcome those boundaries and did not only look at UC systems in an isolated manner but also in their social context of use. For example, David et al. [32] present results from the multidisciplinary research cluster VENUS\(^2\). The

---

\(^2\)From January 2010 until December 2013, VENUS was a research cluster at the multidisciplinary research centre for information system design (ITeG) at the University of Kassel. VENUS is a German pseudo acronym and stands for “Gestaltung technisch-sozialer Vernetzung in situativen ubiquitären Systemen”. The research project was funded as a part of the research programme “LOEWE – Landes-Offensive zur Entwicklung Wissenschaftlich-ökonomischer Exzellenz”. For further information visit http://www.uni-kassel.de/eecs/en/iteg/venus (visited on 26/03/2014).

2.2 Socio-technical Computing Systems
outcome of this initiative is a multidisciplinary development method for socio-technical system design that inherently integrates the different stakeholders to overcome the disciplinary boundaries.

We participated in the VENUS project, especially within the sub-project Adaptation as well as in the development of the multidisciplinary software development method. This work contributes to socio-technical system development as it allows systematic user participation in adaptive (ubiquitous computing) software to respect the user’s social context at run-time in an improved manner.

2.3 Context-aware Computing

To define our understanding of Context-aware Computing we first need to specify the term context. Context (from Latin contextus connection of words, coherence, from contexere to weave together, from com- + texere to weave)$^3$ describes the conditions and circumstances of an entity.

One of the most widely referenced definitions of context in the area of computing is given by Abowd et al. [2]. They refer to context as “any information that can be used to characterise the situation of entities (i.e. whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves” [2].

We will use the definition from Wagner [154] who adopted and revised the above definition from Abowd et al. to better fit latest developments:

“Context is the information, which can be used to characterize the situation of an entity. Entities are persons, locations, or objects which are considered to be relevant for the behaviour of an application. The entity itself is regarded as part of its context.” [154]

Context Awareness is a property of software, which considers the current context of the system or the user (i.e. the entity) to adjust the provided functionality. Abowd et al. [2] describe context-aware systems as systems that use context to provide the user with relevant information or services. Huebscher and McCann also include system adaptivity in their definition:

“Context-awareness is the ability of an application to adapt itself to the context of its user(s).” [191]

Wagner integrates these two perspectives on context awareness into this more general definition:

“An application is context-aware, if its behaviour is influenced by information on its context.” [154]

In the next section, we will discuss self-adaptive systems (cf. Section 2.4.1). Compared to context-aware systems, self-adaptive systems are specialised on systematic adaptation aspects (based on context) and system automation without user involvement. Elaborated mechanisms have been developed to achieve software re-configuration during run-time.

Hence, we can see context awareness as the more general concept which is also used by self-adaptive systems.

2.4 Adaptive Software Systems

An essential part of ubiquitous computing systems is adaptation to changing run-time conditions in dynamic execution environments. Depending on the field of application and research background, adaptation can have different characteristics. In the following, we will present three different types of adaptive systems: self-adaptive systems, adaptive user interfaces, and socio-adaptive systems.

2.4.1 Autonomic Computing and Self-Adaptive Systems

Intelligence and anticipation of UC systems typically involve system adaptation to changing run-time conditions, for example to better fit the current context or to optimise available system resources and provided service quality. These automation and adaptation tasks make the development of software costly and cumbersome.

Autonomic Computing (AC) [106] addresses the increasing complexity of software systems with the goal to reduce maintenance and management effort by humans. The system itself should take responsibility for configuration, adaptation, healing, or optimisation. Such properties are called self*-properties and software systems can be, e.g. self-adaptive, self-healing, or self-optimising. AC systematically shifts the management of complexity from humans to software [84].

Autonomic computing is a means to automate human tasks that are simple and repeated often. However, the idea of AC goes beyond simple automation of tasks. Such systems are capable to autonomously deal with changes in their environment, i.e. their context, and making decisions depending on the situation. It is important to understand the difference between automation and autonomous decision making. AC is the more generally term and typically involves automation. An autonomic computing system would always decide in the same manner in the same situation. Hence, from a user’s perspective, this may look like automation.

However, considering AC systems as systems that automate particular tasks is helping us to analyse the human-computer interaction part. Automation is a well-known topic in human-computer interaction research [80, 111, 130, 133] originated from research on control systems. For example, Norman claims that automated control systems gave inappropriate or no feedback to their operators making it hard for the user to understand its actions [111]. Another issue is Automation Surprise [130] which always occurs when the (automated) behaviour of the real system diverges from the mental model of a human user. A user builds upon his mental model by prior information on the software system and while actually using it. The concrete interactions with the system are then guided by the mental model [71]. The combination of lack of feedback and automation surprise reinforces the need for a user interaction concept in automated systems.

However, one can argue that the idea of AC is to transfer responsibility from the user to the system and as the term autonomic indicates that no interaction between user and
system is intended. We do not support this argumentation for two reasons: first, only few computing systems require no interaction at all and second, even if there is no or only little interaction with the system, the system may influence the human with its actions. Such action can be either directly related to the human or the systems alters the environment. Although working autonomously, we cannot neglect the interplay between humans and autonomous software systems.

Klein et al. [80] name four basic requirements for automation in joint human-agent activity in the robotic domain while an agent represents an autonomous software system. Such requirements are:

1. Achieve a common agreement on shared goals and beliefs.
2. Be mutually predictable.
3. Be mutually directable (i.e., controllable).
4. Establish common ground.

In this work, particularly the second and third requirement will be addressed. The first and last requirement refers to world modelling and reasoning, which are assumed to be available.

Sheridan and Parasuraman [134] define eight degrees of automation in interactive computing systems, i.e., systems that involve user interaction (cf. Table 2.1). The scale starts with a system in which it offers no assistance at all and it ends with an autonomic computing system in which the user is completely ignored. In Chapter 3 we will explain why it might be meaningful to have a system that dynamically addresses a range of automation degrees depending on the situation and the user's preferences.

<table>
<thead>
<tr>
<th>Table 2.1: Degrees of automation according to Sheridan and Parasuraman [134].</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The computer offers no assistance; the human must do it all.</td>
</tr>
<tr>
<td>2. The computer suggests alternative ways to do the task.</td>
</tr>
<tr>
<td>3. The computer selects one way to do the task and</td>
</tr>
<tr>
<td>4. executes that suggestion if the human approves, or</td>
</tr>
<tr>
<td>5. allows the human a restricted time to veto before automatic execution, or</td>
</tr>
<tr>
<td>6. executes the suggestion automatically, then necessarily informs the human, or</td>
</tr>
<tr>
<td>7. executes the suggestion automatically, then informs the human only if asked.</td>
</tr>
<tr>
<td>8. The computer selects the method, executes the task, and ignores the human.</td>
</tr>
</tbody>
</table>

Self-adaptation is an essential property of autonomic computing. It refers to a system's capability to dynamic reconfigure aspects of its own software at run-time [27, 194]. This is necessary to allow the system adjustments to new situations. A self-adaptive system can adapt its behaviour and structure in response to its perception of the environment. The prefix self refers to the autonomic computing origin as the system needs to determine on its own when to adapt and how to adapt. We will describe the details of self-adaptation in Chapter 3.

Traditional software is static; it is intended for a particular purpose under very constrained conditions. Although many software systems can be adjusted through parameters by
users or administrators, they are not able to react on its own on changing environmental conditions. The development of self-adaptive software is significantly more challenging than static and predictable software, in particular when dealing with uncertainty [27]. Self-adaptation promises more versatile, flexible, resilient, dependable, energy-efficient, recoverable, customizable, and configurable software systems [90]. For example, on resource-constraint mobile devices an elaborate self-adaptive application might perfectly adjust to changing run-time conditions, limited memory capacities, or the integration with other software systems. Unlike context awareness, self-adaptation not only allows software to consider context changes but it also measures its own performance and can decide on its own how to react on the changes and it can autonomously perform changes in the programming model. Self-adaptation is not originated in the field of ubiquitous computing. Frankly, a lot of research is done in other fields like Service-oriented Computing (SOC). In SOC, systems automatically adjust their Quality of Service (QoS) parameters depending on changes in the context, for example available network bandwidth, number of users, and so on. Lately, researchers notice a paradigm shift [40] from QoS optimisation systems to more user-centred self-adaptive systems [141] as in UC.

Although we discuss the properties of self-adaptive software systems more detailed in Chapter 3, we refer to the most recent research studies on self-adaptive software systems [72, 92, 128, 162, 163].

2.4.2 Adaptive User Interfaces

Unlike autonomic computing, Adaptive User Interface (AUI) research explicitly targets on interactive applications with user involvement. In fact, the idea is to improve usability through user interface adaptation [18]. Compared to autonomic computing this is somehow the opposite motivation as autonomic computing faces the problem of decreased usability through autonomous self-adaptive behaviour. However, adaptation in AUI systems is much more limited as there is usually no autonomous reasoning process that decides about adaptation. Further, only the user interface is considered for adaptation, all other components of the system remain static and predictable.

The target group of AUI systems are users with special needs, for example disabled [143] or elderly people [36]. Through dynamic user interface adaptation those users are able to interact with computing systems in the same way as people without handicap could do. With AUI, software developers do not need to provide completely independent versions of their software depending on the target group, but rather the same software automatically adjusts to the user.

Another application of AUI systems are multimodal user interface [33]. Systems can provide different types of user interfaces like voice, gesture, or touch interfaces. Depending on the context, an AUI system can select the best interface for the current situation.

Although focusing only on the user interface, AUI systems face similar problems as self-adaptive systems, notably transparency and controllability [86, 115]. However, AUI adaptation does not affect the actual software architecture and adaptation can be on a quite low level like changing font colours or font sizes. Hence, adaptation complexity is limited to the user interface.
2.4.3 Socio-Adaptive Systems

A new class of adaptive systems are so-called Socio-Adaptive Systems, a term recently defined by researchers at Carnegie Mellon Software Engineering Institute in the context of Ultra-Large-Scale Systems (ULSS)\(^4\). They define socio-adaptive systems as follows:

“Socio-Adaptive Systems are systems in which human and computational elements interact as peers. The behavior of the system arises from the properties of both types of elements and the nature of how they collectively react to changes in their environment including mission, and the availability of the resources they use.” [60]

Unlike autonomic computing and adaptive user interfaces, socio-adaptive systems are considered to be a socio-technical system by definition. Users and generally groups of stakeholders are an essential component of such systems. The main problem described in the technical report [60] from Carnegie Mellon Software Engineering Institute, is the mission-aware allocation of resources to users. Their primary example are military systems, i.e. highly distributed systems involving humans and “computational elements” in an uncertain environment. The allocation of scarce network resources in combination with QoS optimisation is mentioned as the main challenge.

The term socio-adaptive system seems to be rather new and according to the above definition it must be said that such systems do not focus on classical “socio” attributes like trust, privacy, usability, or social group dynamics. They rather consider multi-objective run-time resource allocation from different stakeholders. The system has to adapt to the changing requests from the stakeholders. A user does not have to cope and maybe struggle with the dynamics of the adaptive system.

2.5 Human-Computer Interaction

Human-Computer Interaction (HCI) is an important factor for the design of ubiquitous computing systems. People need to interact with software to accomplish their current tasks. HCI wants to ease the interaction between humans and computing machines while simultaneously optimising the efficiency of interaction. Interaction design [132] goes beyond classical desktop computing systems. Challenges occur within mobile and ubiquitous computing where people are confronted with new types of interaction. Interaction design keeps in mind Weiser’s vision that computing devices move in the background of everyday computing. Modern types of interaction like voice input, gestures, touch screens and touch sensors, or RFID-tag-based interfaces provide new chances but also challenges for the design of systems and applications.

When we talk about the integration of user participation in adaptive software we surely have to consider the way the users are interacting with such systems. Important aspects are the usability in general and specific adaptation-related problems like interruption, intrusiveness, or obtrusiveness. In addition, users need to adjust the system to their personal individual preferences. We will discuss these topics in the following sections.

---

\(^4\)The term ULSS describes software-intensive systems with large amounts of hardware, software, involved people, and data. It is argued that traditional engineering approaches do not work for those systems.
2.5.1 Usability

Usability is an important quality attribute for human-made objects and software in particular. Simply speaking, it describes the ease of use and learnability of interaction. Several international standards deal with usability in general and with usability in interactive systems [66, 69, 70].

ISO 9241-11:1998 [66] on ergonomic requirements for office work with visual display terminals defines usability as

“the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.” [66]

The product refers to an adaptive application or system for example. The context of use is composed of the user, his tasks he wants to accomplish, the means for work beside the system and the physical and social characteristic of the environment [66]. In order to design a usable application its design has to consider the characteristics of each element, for instance the age, education and special needs of the users, the frequency of the tasks, the user wants to fulfil with the application, other environmental information or social etiquette of the environment in which the application is used. Further, it is important to analyse existing applications and systems within the same or a similar context of use. ISO 9241-210:2010 standardises ergonomics of human-system interaction and provides guidance on how to understand and explain the context of use for human-centred design in interactive systems [69].

Further, ISO/TR 16982:2002 [70] proposes methods on how to achieve usability in interactive systems. The focus of those design methods is to determine user’s knowledge, capabilities, and limitations relative to the task for which the system is being designed. Examples are observation of users, questionnaires, interviews, thinking aloud, or expert evaluation.

Unlike in self-adaptive systems, usability claims that the user should be in control at any time. Especially ISO 9241-110:2006 [67] includes the basic dialogue principle controllability. Controllability defines that the user should always have the option to start a dialogue, to manipulate the dialogue’s direction, and its pace. For this work, usability is a secondary precondition and goal, but to achieve user participation in adaptive systems, we must not neglect the system’s usability at the expense of efficiency, flexibility, or reliability. An important aspect is the so-called context of use. For adaptive systems, we have to consider the adaptive behaviour in the context of use. As we propose a bi-directional communication between the adaptive system and the user, we explicitly establish human-computer interaction for adaptive systems. Hence, we need to keep in mind usability requirements while designing the human-computer interface.

Beyond international standards, accepted guidelines like Shneiderman’s eight golden rules for usable interface design support designers during application development [135].

2.5.2 User Interruption

Ubiquitous computing systems claim for the user’s attention. As UC shifts the focus from operating a single computing system to multiple ubiquitous systems at the same time,
users are confronted with many systems concurrently. For example, when living in a smart home environment, a user has to interact with the smart home system from time to time while at the same time using his smart phone or smart television device to achieve other goals.

To direct the user's focus to one of the systems if required, interruptions are necessary. Generally, McFarlane [97] defines (human) interruption in the context of HCI as follows:

“Human interruption is the process of coordinating abrupt change in people's activities.” [97]

Hence, we can imply that interruptions are initiated by the computing system. We distinct two types of interruption: first, systems interrupt to get information from the user or to notify the user with the goal to make progress in any way. And second, autonomous and adaptive systems interrupt to adapt themselves to better fit the current situations. Then the current adaptation can be seen as an interruption. Depending on the degree of adaptation, such adaptations can be an abrupt change in the user's current activity. The latter is usually unwanted in any case and needs to be avoided. However, there is no known research how to avoid interruptions by adaptations. According to Kniewel et al. adaptation creates divided attention and multitasking situations [223].

For the former type, interruption is a necessary means in HCI and whenever there is a need for communication between the user and the system. However, the required user attention is a limited resource of human-computer interaction [63]. Moreover, interruption can mediate disruption from the user's current task [9]. Other terms used in this context are intrusion [122] or obtrusion [51]. They all refer to interruptions as the major cause. Interruptions usually imply a distraction from the current task and it is difficult to resume this pre-interruption task after a post-interruption task had to be handled. The goal is to minimise errors made by the user due to interruptions.

According to Cellier and Eyrolle [23] there is a useful distinction of errors made by users related to interruption. They observed three subgroups of specific errors: intrusions, confusions and omissions: “intrusions are errors where people incorrectly perform actions for the pre-interruption task after task-switching. Confusions are errors where people accidentally mix actions from pre- and post-interruption tasks. Omissions are errors where people fail to perform part of the post-interruption task” [23].

To coordinate interruption McFarlane [96] presents a taxonomy with four known ways: immediate, negotiated, mediated, or scheduled. With immediate coordination, the user has to react immediately on the post-interruption task. One problem with this design solution is that people experience a so-called automation deficit when they resume interrupted tasks, i.e. the performance on the pre-interruption task decreases. The negotiated solution addresses people's natural ability to deal with changes in their activities. Clark [30] found four possible responses to interruptions in normal human language (i.e. messages):

- take-up with full compliance
- take-up with alteration
- decline
- withdraw
Compared to immediate coordination, people now have the choice how to react on the interruption. However, there can be overhead (coordination) costs for the user and people may prefer immediate interruption if the overhead cost is not justified.

Mediated coordination delegates the interruption problem to a mediator which can be an intelligent user interface that signals interruption to the user. It can also be an algorithm that predicts the users’ interruptibility or that calculates the user’s cognitive workload for dynamic task allocation. However, in any case the user has to supervise the mediator which generates a new task.

Scheduled coordination transforms sudden interruptions into normally planned activities. According to McFarlane, “time management training has been found to have a positive effect on people’s ability to manage interruptions” [96].

McFarlane summarises in his analysis on interruption coordination that there is no best choice of method for coordinating interruptions. Every coordination method imposes a tradeoff that designers have to keep in mind. However, he also provides a few guidelines on how the different methods can be used [96]:

1. Negotiated solution is best and scheduled solution is worst for accuracy on a continuous task.
2. Scheduled solution is best and immediate solution is worst for causing fewest task switches.
3. Immediate solution is worst for accuracy on an intermittent task.
4. Immediate or mediated solutions are best for completeness on an intermittent task.
5. Immediate solution is best for promptness on an intermittent task.

Horvitz and Apacible [64] introduce cost of interruption with the goal to determine when it is best to ask, i.e. to interrupt the user. They use machine learning techniques to classify the user's current attentive state. Several research works try to reduce interruptions by using context-aware computing. For example, Ho and Intille [61] studied when it is best to send messages to the user of a mobile device. They suggest sending messages pro-actively when a user is transitioning between two physical activities like sitting or walking. Fogarty et al. [44] use a sensor-equipment environment to predict when users can be interrupted least or best.

On the contrary, in mobile ubiquitous computing systems, interruption by the system is a problem that has been examined by several researchers [77, 124, 148]. In static and predictive mobile systems, notifications are the main cause for interruptions.

When we want to achieve user participation in adaptive software systems, we have to keep in mind the effects interruptions might have on the user, especially the ones that disturb the user. For the design of suitable participation mechanisms we also have to deliberate about the different types of user reaction on interruptions, depending on the current task.

### 2.5.3 Trust

The formation of trust is a central issue in the design of UC systems and it was only a minor issue in traditional (desktop) computing systems. There are several reasons. For
example, UC systems often have new types of user interfaces that do not look familiar. Further, context-aware as well as self-adaptive applications derive their behaviour from complex machine learning algorithms or semantic interference models. The user may not understand why the system behaves in a certain way. As UC systems are highly integrated in people's everyday life they can also be much more intimidating, especially if they rely on personal sensor information or sensitive private data. Hence, the willingness to trust such systems can be lower than in traditional computing systems.

Research focuses on initial also known as short term trust [8, 139], i.e. trust that is formed after first experiences and interactions with the computing system. Initial trust is important for the general acceptance of the system and all further examinations on trust will rely on the initial trust.

We follow the argumentation from Söllner et al. [139]. They analysed the formation of trust in IT artefacts based on the concept of trust in automation from the HCI discipline rather than trust research in human-to-human communication. They argue that “IT artefacts cannot be compared to human beings in a way necessary for relying on the foundation of interpersonal trust” [139]. Hence, Söllner et al. use the three dimensions performance, process, and purpose from Lee and Moray [88] to assess trust. The performance dimension reflects the ability of the system to help the user achieving his goal. In contrast, purpose describes the user’s perceptions of the intentions the system’s designers had and the future value of the system. The process dimension represents the user’s comprehension of the algorithms and processes the systems comprise.

In a laboratory experiment with the Meet-U application\(^5\), Söllner et al. measured the impact of the three dimensions on the formation of trust. For each dimension, they assigned up to five formative indicators that were evaluated with a questionnaire, e.g. reliability (performance), understandability (process), or authorised data usage (purpose). The impact is statistically significant for all dimensions with process and purpose dimensions having the most impact on trust formation in Meet-U.

Although Söllner et al. show which dimensions affect trust, they do not say anything about how to achieve trust in practice. While in the past there has been trust research in automation and expert systems Muir [103], current work on context-aware and adaptive systems focuses on concrete properties of systems like intelligibility, transparency, or accountability [6, 8, 45, 91, 150]. Often researchers assume that these properties positively affect trust. Indeed, we can assign intelligibility and transparency to the process dimensions while accountability matches to the performance dimension. However, only little attention is paid to the purpose dimension.

Important aspects to facilitate intelligibility, transparency, or predictability are mental models [109]. Norman [110] further introduces the terms Gulf of Evaluation and Gulf of Execution. While the former describes the separation between the perceived functionality of the system and the user’s intentions and expectations, the latter is the separation between what can be done with the system and the user’s perception of that. The mental model build by the user must match the system behaviour and the designer’s intention. Lack of intelligibility and predictability directly refers to a mismatch between user expectation and system behaviour. Mental models of users are not static, they change over time and designers must support the creation of mental models that are

---

\(^5\)For a description of the Meet-U application see Chapter 12
precise and predictable to gain the user’s trust [150]. Muir supports this argument for automation systems: “[…] trust comes from an ability to predict the system's behaviour through observations” [103].

To support the creation of a stable and correct mental model users need high-level feedback from the system to adopt correct structures and it is not enough to provide only simple feature of the internal reasoning mechanisms [150]. Beside explicit feedback the predictability of a system depends on how easy the system is to observe [8]. Fong et al. [45] define intelligible as context-aware application's ability to reveal its inner workings to users”. This is a quite general assumption as not all information may be relevant to build a proper mental model. In fact, revealing too much or too technical information can further confuse users.

Antifakos et al. [8] address the unreliability of sensor information in context-aware systems. They propose to reveal the current degree of reliability to users so that they can better react on the system's behaviour. They found out that people would less check on the systems correct functionality if the system tells the user that it is confident about its own actions.

Lim et al. [91] examined how explanations can improve intelligibility of context-aware systems. In a controlled lab study they found out that explaining why a system behaves in a certain way and why it did not behave in a different way provided most benefit in terms of objective understanding and feelings of trust compared to other types of intelligibility. The explanations should allow users to build-up a suitable mental model for the particular system. However, Lim et al. noticed that prior knowledge for the task can hinder build up a suitable mental model. In this case they suggest that explanations should help users to understand why a system behaves differently to their everyday understanding. They also see two problems with real context-aware applications: first, generating explanations for complex decision making engines is hard or even impossible (for black box approaches) and second, users may not like explanations all the time but rather on demand as they might be to obtrusive else (cf. Section 2.5.2).

Fong et al. [45] follow the work by Lim et al. and present a framework to support intelligibility in pervasive applications. The framework identifies and exposes adaptation models and transforms them into understandable representation for non-technical users. They externalise situations and preferences evaluations and generate explanations for the decision for different levels of user expertise.

As we described before, accountability directly refers to trust in the process dimensions. Anderson et al. [6] describe accountability as “[…] a primary requirement for making sense of and trusting system behaviour.” Accountable systems can provide explanations “why things are as they are” for the respective purpose. The purpose aspect is the main difference over intelligibility. Users need to make sense of the system’s intentions. Bellotti and Edwards [17] state: “context-aware systems must be accountable and so must their users. Users need to be able to understand how a system is interpreting the state of the world” [17]. However, it is not sufficient and productive to provide all available information to the user but rather this information need to be represented in a way from which the user can account for the system’s behaviour: “[…] the availability of contextual information does not ameliorate the problem of providing accounts of system behaviour that the user can understand and trust” [6].
To increase system accountability, intelligibility, transparency, predictability, and hence trust, the majority of approaches let the system provide additional information to adjust the user's mental model. To achieve trust by providing active user participation, i.e. restoring user control in context-aware and adaptive systems is not considered. Although reduced controllability is a significant difference between static and adaptive systems, Söllner et al. included trust by controllability in their experiment. It can be assumed that lack of control directly refers to a lack of trust. Kaminski et al. say that “Pre-autonomic systems had a relatively direct connection between the users' commands and the actions taken by the software” and that “the system no longer exhibits a direct cause-and-effect relationship between commands and actions” [73]. However, in adaptive user interface research, controllability is an important research topic [86] and so-called mixed-initiative interfaces focus on the collaboration between intelligent interfaces and their users [63]. We argue in Bellotti and Edwards' favour who want to “provide control […] to the user, over system and other user actions that impact him or her, especially in cases of conflict of interest” [17].

### 2.5.4 Individualisation

Individualisation in UC systems refers to customisation or personalisation, i.e. adjusting the systems to the user's personal needs. The shift away from desktop systems to ubiquitous computing brings some new challenges related to individualisation. First, there is a move from the quiet office environment to an everywhere use of systems and software. Software is not anymore restricted to be used in one place under very specific environmental conditions. Depending on the context, each user may have different requirements and preferences towards the system. And second, computing systems have become much more mobile and heterogeneous. This involves a change in usage patterns for every single device. Also, limitations from the hardware could be addressed differently by every user. Generally, UC systems are often context-aware and they make use of many personal preferences to support the user in fulfilling his tasks. Consequently, users need to configure and adjust those preferences. Hence, it is not surprising that personalising context-aware applications is considered in many publications [34, 57, 58, 81, 138].

ISO 9241-110:2006 [67] on dialogue principles in human-system interaction lists individualisation as one of the dialogue criteria and defines individualisation as follows:

“A dialogue can be individualised if the system allows modification of the human-computer interaction and the presentation of information to the individual abilities and preferences of the user.” [67]

The standardisation document suggests to use individualisation carefully and not as an excuse for badly designed user interfaces. However, it can be an appropriate solution to lower barriers for accessibility, e.g. elderly people.

ISO 9241-129:2010 [68] extends ISO 9241-110:2006 and provides guidance on software individualisation including modern interactive systems as used in UC:

“Individualisation comprises modification regarding the behaviour of an interactive system and the presentation of user interface elements before
or during actual system usage to better conform to attributes of the current context of use for single or multiple users.

[...]

Individualisation refers to modifications that can be done without making changes in the software's program code as the capabilities for individualisation are already integrated within the software.” [68]

Compared to ISO 9241-110:2006 (dialogue principles), the newer ISO 9241-129:2010 on software individualisation includes a behavioural dimension and does not anymore explicitly discourage the use of individualisation in interactive systems. Moreover, the moment of time when users are able to make changes addresses the offline and online condition. It further takes into account the need for support of multiple users, i.e. different users may have different requirements or a single user has different requirements at different times. Part 129 of ISO 9241 is not anymore limited to traditional interactive systems and particularly addresses systems in the ubiquitous computing domain.

ISO 9241-129:2010 explicitly describes the following types of individualisation: user settings, user profiles, interface skins, recurring interaction tasks (automation), and content customisation (filtering, sorting, or searching).

Further, part 129 suggests providing individualisation in the following cases:

- Variation in user capabilities: increase accessibility between different user groups or different users within a group. It can also be used when those capabilities change permanently due to new qualifications or medical conditions.
- Different goals and requirements of users: users that use the system less frequently might need additional explanation that can be enabled at run-time.
- Changes in task attributes of users: the tasks associated to users may vary in complexity, difficulty, or intensity. The system should be adjustable to the task attributes.
- Changing infrastructure: the system should be adaptable to the available infrastructure, e.g. different input devices like tablet computers, desktop computers, or mobile phones.
- Varying environments: the system is not necessarily static and may move in different environments implying different environmental conditions, too, e.g. a mobile computing system. The system should be adaptable to those different conditions.

Configuring systems involves human-computer interaction. The standard ISO 9241-129:2010 points out that people will need interfaces to implement their personal preferences. However, the degree of individualisation is manifold. It starts with simple settings dialogues as indicated in ISO 9241-129:2010 and ends with elaborate end-user programming allowing users to build their own applications according to particular rules.

It is not only about changing preferences of applications on the dialogue level, it is also about the internal representation of those. For example, Poladian et al. [118] developed a formalism to represent user need and preferences and a mechanism to exploit such representation for automatically configuring intelligent environments.

2.5 Human-Computer Interaction
Henricksen et al. [58] provide a generic preferences abstraction model for developers. Such preferences are used to drive a context-dependent behaviour of applications. The preferences and the related context model are specified externally to the application. The authors imply easy manipulation and evolving of the models without changing the source code. Preferences are basically modelled as policies, i.e. rules by the developer. It is claimed that the provision of such high-level abstractions supports the development of context-aware applications.

Unlike Henricksen et al., Fong et al. [46] present a user-centric preference modelling approach that aims at explanations of preferences and customisation of the decision-making process based on rules. Each user preference is composed of a set of rules that helps to externalise the preference modelling of adaptive applications.

The next step is intelligent individualisation, especially in mobile applications on portable devices. Intelligent typically means the adoption of machine learning techniques to learn individual preferences of users in the course of time. For example, Gil et al. [51] propose personalisation for unobtrusive service interaction while interaction refers to notifications from ubiquitous computing systems. They use feature models to decompose the different context conditions according to which they associate levels of obtrusiveness. They use personas\(^6\) to describe a user more precisely and hence allow a personalised notification from services with minimum interruption (cf. Section 2.5.2).

Valtonen et al. [151] developed adaptive user profiles with fuzzy logic (cf. Chapter 6) for mobile phones. Interestingly, the fuzzy rules are created by the system itself and not by the user, i.e. the fuzzy control automatically adapts to new user situations based on observations. However, users can override settings or suspend automatic reasoning.

Google Now\(^7\) collects information about a user to make meaningful predictions and to anticipate the user's moves in advance. It achieves individualisation by observing the user and by learning on those observations. Google Now supports the user with additional information according to the learned user behaviour. For the user there is no need to configure any preferences or rules.

While mobile phone software needs to be personalised for changes in infrastructure and the execution environment, other applications like Smart Homes focus on multiple and diverse users in a more static environment. Particularly for elderly or disabled people a system needs to be adaptable and tailored according to the users' needs [153, 157].

In the field of context-aware computing, interactive prototyping and end-user programming is a popular research topic [34, 81, 138] as it allows ordinary users to build their own context-aware applications. Korpipaa et al. [81] propose smartphone customisation based on a set of available context information and actions. Users can define rules to achieve the desired context-aware behaviour. Korpipaa et al. argue that predefined context-aware features are too rigid. This approach can be seen as lightweight end-user programming because no real re-configuration of software can happen.

---

\(^6\) The development of personas is a method in usability engineering. Personas comprehensively describe a hypothetical archetype of potential users and give clarity to their needs. The concept of personas has also been used during the development of the demonstration application Meet-U (cf. Chapter 12).

\(^7\) Google Now is an integral part of the Google Search application on the Android mobile system platform. Google Search is bundled with many of the latest versions of the Android system, but it can be obtained from the Google Play Store (https://play.google.com/store, visited on 26/03/2014), too.
Similar to the work by Korpipaa et al. is end-user programming to support automation of common tasks which already exists in practice. Mobile applications like Smartactions, Tasker, or Automagic for the Android mobile system platform support the user in task automation. However, they do not provide real end-user programming and mostly use system-defined triggers to automate tasks like muting the phone, enabling or disabling Wi-Fi, or sending short messages at specific locations. This can be described as user customisation on system level and not on a more specific application level. Some of the used methods are quite advanced, e.g. Automagic adopts a flow chart modelling technique to describe automation processes.

Kniewel et al. [223] exposed in a user study on adaptive applications that it is required to provide a preference settings dialogue where the user may adjust the adaptive behaviour of the adaptive application. Even if the user will not make use of changing the settings, he clearly wants to be aware of the option. Kniewel et al. propose a combination of settings profiles and a construction kit to support novice user with behaviour management of adaptive applications.

On the contrary, Barkhuus and Dey [14] earlier found out that people prefer context-aware features over personalisation, but in the same moment they perceive a lack of control. Thus, intelligent anticipation by the system is appreciated over manual configuration and users are willing to accept a loss of control as long as the application’s usefulness is greater than the cost of limited control.

2.6 Summary

Ubiquitous computing systems have been developed for many different directions and application scenarios. In this work we focus on socio-technical and dynamic systems in heterogeneous environments that require or desire user interaction. Further, we consider systems that dynamically adapt and re-configure themselves to new situations. Such systems are not only context-aware but rather they are self-adaptive. Typical application scenarios are smart homes, mobile phones, or intelligent vehicles.

UC systems reveal many chances for future computing systems and their users, but also bear risks if not addressed appropriately, particularly if the technical solutions divert from user expectations and requirements. Such risks can be insufficient usability, interrupting notifications and adaptations, decreased controllability and consequently decreased trust in those systems. Other risks are security and privacy concerns which are only addressed indirectly by this work, for example by increasing transparency and control. Generally speaking, there is a risk of putting the user out of the loop.

User interaction and usability and hence dialogue properties of software are rooted in the field of human-computer interaction. Trust research is slightly more interdisciplinary, involving psychologists, interaction designers, ergonomists, or economists. However, we will focus on the technical perspective of user participation, i.e. how we can modify and extend closed-loop self-adaptive system to enable (improved) user participation. We leave trust [139], interaction design and usability [223], and legal [224] research in UC systems to our colleagues from project VENUS.

8The applications Smartactions, Tasker, or Automagic for the Google Android mobile system platform can be found in the Google Play Store (https://play.google.com/store, visited on 26/03/2014).
3 Self-Adaptive Software

In this chapter we provide the basic foundations for the self-adaptation concept in software systems. Starting with a broad definition and ending with limitations, we also explain some of the essential mechanisms of self-adaptive software (SAS) to identify where user participation is reasonable and possible.

Figure 3.1 presents the high-level view on self-adaptive software with two levels describing the SAS. The management-level comprises the self-adaptation mechanisms to modify the actual application. Typically, the management-level is realised as a middleware. The application-level describes the adaptive software or application whose components or parameters are adapted by the software described in the management-level. Adaptation may happen in regard to changes in the environmental context or the software itself. Sensors grasp the environment and the current state of the software system while effectors return feedback in terms of reconfiguration to the software. In this traditional view, humans adopt different roles: first, humans are users, operators, or administrator of the software. And second, the specific behaviour with its objectives needs to be determined by a domain expert or developer prior use.

Although the view from Figure 3.1 dominates the research field, self-adaptation is manifold and there is already much degree of freedom in this particular view. For example, each of the elements in the management-level justifies its own research domain. Moreover, the way how application-level and management-level interact creates new challenges. The following sections will provide a more extensive picture – always having in mind the need for more user participation.
3.1 Definition and Goals

Embedded within autonomic computing, the self-adaptation concept is strongly influenced by control theory, software engineering, artificial intelligence, and distributed computing and hence applied in many fields of science and engineering. For this work we need to define self-adaptation in the context of software systems.

One goal of self-adaptive software is to deal with increasing complexity of software from the management and maintenance perspective. For example, in ubiquitous computing environments or other embedded software applications, software needs to be robust and should adjust to new or even unexpected situations. The increased heterogeneity of the involved components as well as frequent changes in their objectives demand for more dynamic software systems. Particularly those that comply with these requirements during run-time.

Self-adaptation is one of the self-* properties [75] that help to overcome the above challenges. Although not part of the original proposal for autonomic computing from Kephart and Chess [75], it turned out to be an integral mechanism for self-managing, self-organising, self-optimising, self-healing, self-protective, or self-configuring systems.

One of the first definitions of self-adaptive software is from Laddaga [83]:

“Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible” [83]

Oreizy et al. [199] extended the definition from Laddaga by introducing the operating environment:

“Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation.” [199]

Salehie and Tahvildari [128] developed a much more precise definition which also correlates well with our high-level perspective presented in Figure 3.1:

“Self-adaptive software aims to adjust various artifacts or attributes in response to changes in the self and in the context of a software system. By self, we mean the whole body of the software, mostly implemented in several layers, while the context encompasses everything in the operating environment that affects the system’s properties and its behavior. Therefore […] self-adaptive software is a closed-loop system with feedback from the self and the context.” [128]

Salehie and Tahvildari differentiate between context information and information on the system itself. Compared to the previous two definitions they describe SAS systems as closed-loop systems that base their decisions on the self and the context. We will discuss different kinds of feedback systems in Section 3.3.

For this work we will use the definition by Salehie and Tahvildari as it is precise and at the same time general enough to capture all types of SAS for ubiquitous computing systems.
3.2 Taxonomy

There is great variety in the engineering of self-adaptive software systems. Application domains include robotics, mobile applications, sensor networks, embedded systems, system of systems, social services, manufacturing industry, traffic and transportation, or software industry [92]. Solutions for different domains focus on different requirements using different techniques. In this section, we will provide an overview to understand where user participation is possible and to what degree.

Figure 3.2: Taxonomy of the self-adaptation domain (adapted and extended from Salehie and Tahvildari [128]).

Figure 3.2 illustrates a taxonomy of the self-adaptation concept which we have adapted and extended from Salehie and Tahvildari [128]. In addition, Brun et al. [21] define a design space with five clusters (observation, representation, control, identification, and enacting adaptation) to guide developers in the engineering process of SAS. It is guided by the above taxonomy but provides more details on the concrete realisation of SAS. However, the authors do not include potential users at any stage.
The remainder of this chapter provides more details on the different adaptation approaches, reasoning policies, modelling of variability, types of architectural adaptation, uncertainty and the consequences for potential users. The rest of this section briefly describes each top-level element of the above taxonomy.

3.2.1 Adaptation Object

The object of adaptation refers to the what and where to adapt and to what degree it should be adapted. It has three sub-dimensions: layer, artefact & granularity, and impact & cost.

The layer dimension defines where adaptation occurs. This can be either on a middleware or in the application-layer (also known as application-level). Adaptation on a middleware-layer [98] is usually oriented towards QoS optimisation, whereas adaptation on the application-layer is more functional and hence potentially more inflictive on the user. However, application-layer adaptation does not exclude a middleware-based adaptation approach [189]. The application-layer can be adapted by middleware operations, too. Because of a clean software design, middleware-based adaptation is closely connected with component-based software design (i.e. employing a component-connector-based view [146]), computational reflection [93], and the separation of concerns [98].

Artefact & granularity specify the part of the system that is adapted. This can be single aspects, components, the entire architecture, or just methods, parameters, and algorithms. Adaptation can be applied to any part of a software system in fine or coarse levels of granularity.

Regarding the cost and impact dimension, adaptation can be either strong or weak depending on the necessary effort [128]. Changing parameters only is less costly than adapting entire compositions of software components (cf. Section 3.6 on types of adaptation). Analogous, the impact of weaker parameter adaptation is restricted compared to the alteration and integration of completely new software components.

3.2.2 Adaptation Approach

The state of the art in engineering self-adaptive software systems is to employ an architectural representation of the software system with component-based design. In the last years, a second approach from the requirements engineering perspective has been established [5]. The latter usually start with feature models to model variability. While the first one is a white-box approach, also known as architectural adaptation [189, 194, 199, 203, 207], the second is a black-box approach that makes abstractions from the concrete software implementation and does not require any knowledge on the internal structure of the software [13, 186, 165].

We say an adaptation approach is static when the adaptive behaviour is hard-coded at design time of the software (e.g. by decision trees). In contrast, dynamic approaches define external policies [76] that can be defined and managed at run-time to allow a change in the software's behaviour. With externally defined policies, developers or users may influence the behaviour if they are not confident with it. However, there are solutions like MUSIC [187] that employ dynamic policies, but lack of the proper
configuration interfaces for run-time modification. Providing appropriate user interfaces is a mixed blessing. On the one hand, humans are able to change the behaviour, but on the other hand, adaptation policies usually require expert knowledge and user interface that can capture their complexity.

Realisation of adaptation can be either internally or externally. Internal approaches often have multiple control loops to control adaptation while external approaches use the MAPE-K adaptation loop [162]. We describe internal and external adaptation in combination with control loops in Section 3.3.

The terms making and achieving [144] differentiate between engineering self-adaptation into the software (making) and realising self-adaptation by adaptive learning (achieving). Adaptive has its advantages when an uncertain execution environment can be expected and the concrete adaptive behaviour cannot be predicted at design-time. As the human behaviour is considered to be uncertain (cf. Section 3.7), adaptive learning can be a reasonable concept to better adjust adaptive software to its users.

### 3.2.3 Adaptation Type

The adaptation type dimension has three sub-dimensions open/closed, model-based/model-free, and specific/generic adaptation. Closed adaptation systems do not allow any further changes to the adaptation behaviour and structure once deployed. In contrast, new behaviours, components, alternatives can be introduced in systems using open adaptation. Related terms are anticipated and unanticipated adaptation [193]. Genuine unanticipated adaptation is hard to realise as it means that an adaptive system can deal with adaptations that haven not been foreseen at design-time.

With model-free adaptation, systems do not have or build a model of the environment and/or themselves. An example are specific reinforcement learning mechanisms that choose the system's actions based on rewards only. Model-based adaptation mechanisms use models of the environment and themselves to derive the adaptive behaviour. For more details on modelling variability we refer to Section 3.5.

Specific or generic adaptation refers to the application domain. A generic adaptation mechanism can be used for multiple different domains while the specific mechanisms is made for a single purpose. For example, a mechanism enabling adaptation on mobile devices is considered to be generic, whereas an adaptation mechanism that focuses on network latency optimisation only is specific. We will only consider generic approaches for a wider range of applications and domains.

### 3.2.4 Adaptation Environment

It is important to take into account the environment where the SAS is deployed. This dimension has three sub-dimension application domain, observable information, and degree of uncertainty. The actual application domain significantly determines all other dimensions. Moreover, it defines the information sensors will observe. We have to find out which information can be observed and which information is required. This information needs to be represented internally in a proper way. Sensor information might imply a certain degree of uncertainty which is a topic that got more attention in the
last years. Also, humans introduce a great deal of uncertainty into autonomous systems. The more uncertainty a system is exposed to, the more likely it is that the system will not behave as expected. We will discuss uncertainty in Section 3.7.

### 3.2.5 Temporal Characteristics

The temporal characteristics dimension has two sub-dimensions *reactive/proactive adaptation* and *continuous/adaptive monitoring*. A proactive system tries to predict the moment of adaptation, whereas in reactive adaptation the system responds, i.e., adapts after a change has happened. Predicting adaptation introduces a great amount of uncertainty, but it reacts before something happens and makes the system looking more intelligent. Despite the benefit if the prediction is correct, engineering systems with prediction algorithms is more difficult. There is no difference from a user participation perspective. In both dimensions, the decision is not necessarily the best for the user. However, the majority of solutions rely on reactive adaptations.

*Continuous/adaptive monitoring* refers to sensing and collecting of information. A continuous monitoring process observes every required sensor at all times. Adaptive monitoring observers only a subset of sensors to detect anomalies in the features. All other required sensors will be sensed and analysed once an anomaly has been detected. Adaptive monitoring is a good candidate in resource-constraint application domains like embedded systems.

### 3.2.6 Interaction

Interaction is an important dimension in the context of this work. Salehie and Tahvildari name interaction with humans (sub-dimensions *human involvement* and *trust*) but also inter-machine interaction (sub-dimension *interoperability*).

On the one hand *human involvement* is not desired in autonomic computing [106], but on the other hand, Salehie and Tahvildari stress the quality of human involvement when it comes to expressing expectations, policies, or to observe what's happening in the system: "human involvement is essential and quite valuable for improving the manageability and trustworthiness of self-adaptive software." [128]. We discuss the role of the user in Section 3.8.

Adaptive systems are rarely used in isolated environments but rather they belong to greater hierarchies of systems. Especially in ubiquitous computing, multiple systems compete for scarce resources and the users’ attention. Adaptation needs to be coordinated among systems to achieve *interoperability*. Global adaptation can only be achieved by employing middleware structures as in in distributed systems.

*Trust* in computing systems is an important sub-dimension which we already discussed in a larger extent (cf. Section 2.5.3).
3.3 Adaptation Loop and Adaptation Approach

According to Salehie and Tahvildari, a self-adaptive system always embodies a closed-loop mechanism [128], which is based on the MAPE-K adaptation loop [75]. However, there is more to say about adaptation loops than MAPE-K, which of course we will describe subsequently.

3.3.1 Feedback Control Loops

Adaptation or control loops have their origin in control theory which deals with the behaviour of control systems with inputs. The external input of a system is called reference. The reaction of a control system according to the reference is called output. When the output variable of a system needs to follow a reference over time, the controller manipulates the inputs to a system to obtain the desired effect on system's output.

There are three basic types of control systems [82]: open-loop, feed-forward, and feedback systems (cf. Figure 3.3), whereas feedback systems are also known as closed-loop systems.

An open-loop controller uses the input to directly control the system. For example, when controlling a simple dishwasher, the system executes the predefined washing programme without considering anything else. This type of controller is not suitable for dynamic self-adaptive applications as its run-time behaviour is not adaptive.

**Figure 3.3:** The three basic types of control loops in a control system with a single controller (C) and a single system (S): open-loop, feed-forward, and feedback. The controller adjusts the system according to measurements at the reference, context, output, or user feedback.
A feed-forward controller analyses the input to adjust the system based on the input and predictions. A slightly more intelligent dishwasher could measure the degree of pollution of the dishes in advance and then select an appropriate cleaning programme. The difficulty with feed-forward control is that the effect of the external conditions (here: degree of pollution) on the system must be accurately predicted. In classic control system there must not be any unmeasured external conditions that also influence the system, which else would result in a bad performance of the system. Self-adaptive systems are slightly different; they typically do not react on external conditions that were not considered during development. Instead, a self-adaptive system may analyse the context and predict possible actions. However, it is the lack of proper models that sometimes make adaptive applications perform badly from a user's perspective.

A feedback controller uses the oppositional approach to feed-forward. The system’s output is measured and fed back to the controller who uses the feedback to adjust its controlling behaviour. A second intelligent dishwasher starts with the predefined programme and during a single sub-programme it measures progress or success, for example, the current degree of pollution during cleaning or the humidity while drying. The controller than decides to extend or shorten the current programme – it modified the system. We see that the feedback controller reflects the typical adaptation strategy of self-adaptive software when it comes to the self-awareness part. A self-adaptive system continuously evaluates how it performs against the given objectives and it adjusts itself if the performance can be better.

We can summarise that self-adaptive applications may employ feed-forward and feedback control loops at the same time depending on whether it reacts on the self-awareness or the context part. In other words, self-adaptation in autonomic computing is employed as a closed loop. However, we are especially interested in systems that include significant user interaction.

Sousa [141] proposes feedback control with user feedback (cf. Figure 3.4 a)). He notes that an open control-loop can be implicitly and explicitly closed by users. Implicitly when users react on the system’s output by adjusting their own behaviour to the adaptive system and explicitly by intentionally modifying the system’s decision processing.

The proposal from Sousa can be only applied to open-loop or feed-forward systems, i.e. systems without feedback. User can close such open loops. We claim that there is another concept with user feedback and system feedback simultaneously
(cf. Figure 3.4 b)). This basically means that a user can provide additional control to a system which is already able to adapt itself on its own observations.

The term human in the loop refers to the integration of users in control loops of control systems for various reasons, e.g. to achieve collaboration between humans and software agents [131]. Especially for automated systems, researchers early identified the so-called out-of-the-loop performance problem [38] which refers to the decreased ability of operators to deal with rare manual operations when required.

We require the human in the loop to achieve user participation, controllability, and transparency for better usability and trust in adaptive applications.

### 3.3.2 Adaptation Approach

From the perspective where the adaptation happens, i.e. where the adaptation control loops are implemented, the adaptation processing can be external or internal [128]. Internal approaches intertwine the application and the adaptation logic. External approaches have an external adaptation engine that senses and effects the adaptive application. The self-adaptive software system comprises the adaptation engine and the adaptive application. Both approaches are depicted in Figure 3.5. While the internal approach is conceptually easier, the external approach is versatile and conceptually cleaner. The latter separates adaptation logic from application logic while the former intertwines both. Maintainability, transparency, and scalability are much easier to achieve with external adaptation engines. The external adaptation engines have only one or two control loops compared to multiple loops (one for each adaptation) in the internal approach. Although more complicated to develop, external approaches are preferred for sophisticated adaptations projects. There are also efforts to combine external and internal adaptation into a hybrid approach [185].

**Figure 3.5:** Internal vs. external adaptation approach: in the internal approach the self-adaptive software system is one single software system. In the external approach the SAS is comprised of the adaptation engine and the software that is adapted.

Weyns et al. describe internal adaptations mechanisms as “[…] software that realizes adaptation using tactics and techniques that are integrated within the components...”
of the managing system” [164]. This is a very implicit description of adaptation. Several researchers claim for more visibility of adaptation loops [22, 55, 104]. Many existing software systems already contain adaptation loops implicitly so their argument. According to Müller et al. [104] adaptation loops need to become first-class citizens of architecture, design, and infrastructure support. Hebig et al. [55] or Brun et al. [22] explicitly address multiple control loops within one application. It can be assumed that these demands refer to self-adaptive systems employing an internal adaptation approach.

In a controlled experiment Weyns et al. [164] compared internal vs. external adaptation mechanisms regarding design complexity, fault density, and productivity. The authors summarise that external adaptation loops do not significantly reduce the number of necessary software components. However, they noticed a highly reduced control flow complexity with external approaches, which increase understandability of the design, and can improve maintainability and testability of the system. Also, the fault density for internal adaptation is higher, which probably is due to the increased number of adaptation loops in internal approaches. And very importantly, external approaches realise a separation of concerns which has according to Weyns et al. a positive impact on productivity.

For the systematic integration of the user, both internal and external approaches are equally suited. However, with the external approach we only need to define one or two user interfaces for the two available loops while the internal approach requires as many interfaces as the self-adaptive software has loops. Regarding the results from Weyns et al., there are several other arguments in favour of the external approach like the separation of concerns or the support for multiple depending applications within one adaptive system.

3.3.3 MAPE-K Adaptation Loop

The MAPE-K adaptation loop realises an external adaptation approach to achieve architectural adaptation. It was first mentioned in autonomic computing research by IBM [75]. Since then, MAPE-K has been established as the predominant adaptation pattern for self-adaptive software systems [90, 162]. MAPE-K is an external adaptation reasoning approach with a single control loop. However, it may include feed-forward (context) and feedback (self) to adjust the adaptable application. Hence, we use the term adaptation loop when referring to the MAPE-K concept instead of the widely-used term feedback loop.

The MAPE-K adaptation loop consists of five distinct components (cf. Figure 3.6): Monitoring, Analysis, Planning, Execution, and Knowledge. While the first four components also define the four phases within the control loop, the knowledge component is permanently available to be used by the other four components.

During Monitoring, the adaptation engine may continuously observe the sensor inputs for any significant changes. In case of a change it will trigger the Analysis phase to perform some deeper examination of the received sensor values. During analysis, the autonomic manager (i.e. the adaptation engine) will check if the changes require a new planning of the run-time software configuration. In the Planning phase, algorithms determine the new best configuration regarding the changed sensor inputs. If there is a new plan, the associated configuration will be set in the Execution phase.
Figure 3.6: The MAPE-K adaptation loop as proposed for autonomic computing [75]. The Managed System represents the adaptable software and the Autonomic Manager the adaptation engine in the terms of an external adaptation approach.

The original proposal as depicted in Figure 3.6 makes some assumptions regarding the managed system (i.e. the adaptable software). First, the employed sensors in the adaptable software sense the environment for context information and also for information about itself. Hence, the term sensor is also used for internal software measurements. And second, effectors do not necessarily involve actual actuators (e.g. in a robotic scenario) that react on the sensory input. Moreover, this can also mean a reconfiguration of software.

3.4 Adaptation Reasoning Policies

During adaptation planning, a reasoning policy has to determine the best possible configuration according to the current input. The adaptation reasoning process is the heart of the adaptation manager. Kephart and Das define three types of reasoning policies: action-based, goal-based, and utility-function-based reasoning [76]. It can be said that action-based algorithms are easier to understand and more straightforward to implement. Goal-based and utility-function-based policies are more sophisticated and powerful. However, particularly utility-function-based approaches are also much harder to comprehend by humans.

The different adaptation policies can be explained with a state-action model (cf. Figure 3.7). From a current state $S$ the system moves to a new state $S'$ according to an action $a$. A state describes the current software configuration. Typically, there is more than one possible state $S'$ and the actual path between the states $S$ and $S'$ is defined by the respective policy. In the following, we will use the example from Figure 3.7 to describe the three adaptation policies.

3.4.1 Action-based Reasoning

Action-based reasoning is the simplest type of reasoning. To reach another state, the action $a$ is directly specified. The transition to the next state is only based on the current state $S$. Before making the transition, it is unknown how good the next state $S'$ will be.

Actions are specified as logical rules. Rules take context and system parameters as input and infer the action in an IF-THEN manner. The set of all rules constitutes the rule base.
Whenever a decision has to be made, the rule base is consulted, i.e. system and context parameters are fed as input into the rule base. This process is implemented by a rule engine.

Action-based reasoning with an underlying rule base is considered to be a straightforward approach when it comes to implementation and use. However, there are some drawbacks: first, a developer has to keep in mind all possible states and has to define at least one rule for each of the states. If not, a state cannot be reached. second, depending on the number of parameters and states, the rule base may become very large and hard to manage. And third, depending on the current situation, more than one rule can be applied, i.e. more than one state is considered to be the next state. Conflict resolution for rules is required. Some of the problems are addressed by the rule engines while others remain in practice.

Among the three reasoning policies, action-based approaches are considered to be best comprehensible by humans. Yet, Rosenthal et al. [125] claim that rules are too complex to be set by the user in advance. Instead, they used experience sampling to elicit user’s preferences. Later, the rules will be deducted from the preferences.

We argue in favour of the action-based approach when it comes to user participation. It is straightforward and comprehensible by users and better than no or imprecise adaptation reasoning. The presented user participation in Chapter 11 does not make use of the action-based reasoning. However, the employed fuzzy-reasoning concept is also based on rules understandable by humans.

Examples for action-based reasoning in self-adaptation software are RAINBOW [188], the DigiHome platform [123], or the work by Wang [158].

### 3.4.2 Goal-based Reasoning

Instead of considering the current state like the action policy, the goal policy describes the state that needs to be reached. They either describe the desired state \( S' \) or characteristics...
of such state. Hence, multiple states $S_1', S_2', \text{ or } S_3'$ may conform with the defined goal and any of such states is equally acceptable. Goal-based reasoning cannot cover fine distinctions in preference, a state is either desired or not desired. Many of the goal-based approaches belong to the black-box class.

With goal-based reasoning the system has to plan its actions to reach the goal. The developer specifies the high-level goal state and the low-level planning process has to calculate the required low-level actions to reach this state. Compared to action-based or utility-function-based reasoning, the actual planning algorithm needs to be much more sophisticated. Hence, the goal policy is not first choice in resource-constrained applications or whenever time is an important factor.

While in self-adaptive systems research, goal-driven requirements engineering [26, 99] has got some attention, goal-based reasoning in particular has been neglected so far. However, Salehie and Tahvildari [129] recently presented the Goal-Action-Attribute Model (GAAM) to allow run-time adaptation based on cooperative action selection. Their focus is on the optimisation of non-functional requirements.

Although probably more intuitive and easier to manage for applications developers, we will not consider goal-based reasoning for our user participation concept. We assume that users do not have multiple similar goal states for the same task. Multiple goals may also overwhelm user and due the lack of previous research work, we rather build our participation concept on top of the more advanced utility function policy as described in the next section.

### 3.4.3 Utility-function-based Reasoning

Utility-function-based reasoning can be seen as an extension to goal-based reasoning. It maps each goal state of a system to a real scalar value instead of the binary classification into desirable and undesirable states. However, utility-function-based approaches do not calculate the actual path to the goal.

Utility and the corresponding utility functions are an important concept in economics. Utility functions calculate the satisfaction of customers according to particular goods. They have been first proposed to be used in autonomic computing by Kephart and Das [76] as well as Walsh et al. [156]. The utility describes the usefulness of a software configuration regarding the current context and its own performance (context and self-awareness).

A utility function is a mathematical expression to rate the members of a decision set and rank them afterwards. Each of the members is described by multiple parameters. In self-adaptive systems, the decision set contains all possible application variants. An application variant represents one possible configuration of the software (cf. Section 3.5). The best application variant will be applied during the execution phase. The function assigns a scalar utility value to each of the application variants. Typically, a utility value is encoded and normalised in the interval $[0, 1]$ of the real numbers to allow a relative comparability:

$$utility \ u \in \mathbb{R}; \ 0 \leq u \leq 1.$$  

(3.1)

3.4 Adaptation Reasoning Policies  45
Hence, a utility is a relative estimation of the satisfaction when selecting the application variant. The higher the utility value, the more useful or desirable the particular application is. However, we have to keep in mind the drawback of the normalisation to the interval $[0, 1]$. With a relative estimate we lose the ability to infer the absolute usefulness of a particular variant. We cannot say how well an application variant absolutely is, but only how good it is compared to other evaluated variants.

There are several examples of utility function usage in self-adaptive software systems ([37, 43, 114, 117, 118]). They allow efficient decision-making for a large number of application variants and do not run the risk of rule base explosions as it might be the case using action policies. However, utility functions are difficult to implement and even harder to maintain or alter once implemented [184, 43, 72]. Some adaptation engines like the one developed in the MUSIC project, have a single utility function per application [187]. This function has to cover all possible application variants and demands a skilled application developer. As in MUSIC a utility can be basically realised by arbitrary program code, it is not necessarily a strict mathematical expression. Consequently it happened that application developers reverse engineered action-based reasoning with simple rules as IF-ELSE statements. This can be explained by the inability of utility functions to properly handle state-based or scheduling tasks. State machines are far better suited in these cases.

There have been improvements to the MUSIC solution by Khan [193] who developed a more flexible concept. He defines a utility function at every variation point (for the definition of variation points see Section 3.5) to further allow a more unanticipated adaptation and better scalability. Multiple utility functions break the complexity of a single function but require a more precise adjustment of the single functions, especially when created by different developers.

A utility function defines the adaptive behaviour of an application. To enable user participation, a user needs to influence or alter the function without bothering too much with the underlying math. Inspired by Khan, we will present a multi-function approach in Chapter 11. Each input parameter is modelled with its own sub-utility function and we provide multiple predefined types of functions for ease of use.

### 3.5 Variability

Understanding the structure and the behaviour of a self-adaptive system is crucial for developers and users. Developers have to define appropriate application behaviour and users have to build up their mental in regard to the application’s behaviour. Adaptation is closely connected with software variability. Variability describes the ability of a software system to be adjusted according to a particular context. The software can be configured or tailored exactly to its needs. Whenever the system should be configured differently, e.g. in case of alternative functionality, a so-called variation point denotes the branching point into a different configuration.

Software Product Lines (SPL) [116] capture the complexity of software intensive systems by using the same basis of a software family in different contexts. Each of the forks is a called software product line. Feature modelling techniques [19, 74] are used to represent and manage the software variability.
However, SPL are static and defined at design time with no possibility for run-time adjustments. For example, let us consider a software product family for office applications which includes single programs for word processing, spreadsheet calculation, slide presentation, or drawing. All products belong to the same product family and have a common basis. Depending on the required features, each product has its own product line. A feature model captures the different variants. Each of the SPL can again be a software family with own SPL. For example, the spreadsheet application can be distributed in a basic, a standard, and a professional version. Again, a feature model describes common and exclusive functionality. In all cases, the software is static once compiled and deployed.

Dynamic Software Product Lines (DSPL) extend the classic SPL approach by deploying reconfigurable software products [52]. Morin et al. [101] use Model-driven Development (MDD) and aspect weaving techniques to create highly dynamic run-time models of DSPL. Their main goal is to reduce the number of different configurations that are relevant in each situation. Abbas et al. [1] continued the development in DSPL research to allow for Autonomic Software Product Lines (ASPL). ASPL can re-configure themselves at run-time according to changes in the execution context, e.g. to target group or environment. ASPL can be seen as self-adapting SPL although some DSPL have self-adapting features, too.

SPL, DSPL, and ASPL are conceptually situated in the early requirements analysis phases of the development process and focus on the differentiation between entire software products. They neither capture the fine-grained adaptation requirements of software components nor are they able to model other technical details necessary for adaptation. However, researchers employ feature models to manage run-time variability of adaptive systems (cf. Acher et al. [3]), especially in requirements-driven adaptation and goal-based approaches [84]. However, feature models are per se incapable of providing run-time adaptations to contextual or self changes. Further extensions are required [54].

Achieving run-time variability of component-based software was the goal of the projects MADAM [189] and MUSIC [187]. Both use UML-based variability models to represent application variability. Component-based software allows flexible and modularised software architectures, e.g. for service-oriented computing. Alférez et al. [4] use variability models in web service compositions. The variability model is dynamically created respectively altered at run-time, while in MADAM and MUSIC the variability model is created at design time and valid variants are derived at run-time. Adaptation planning and modelling is also used in the domain of self-adaptive autonomous mobile robots for efficient engineering and dynamic run-time behaviour of those systems [137].

Figure 3.8 depicts two types of variability models of the fictive component-based application HelloWorldApp. It consists of two components, one being the user interface and the other one comprises the application logic to have separation between the graphical user interface (GUI) and the programming model. Depending on the device’s screen orientation the user interface can be either in portrait or in landscape mode. Two different application logic components can be used depending on the scenario. The corresponding feature model is described in Figure 3.8 a). The two variation points in

Component-based software engineering focuses on the separation of concerns within software engineering. Independent and loosely coupled components can be reused and provide the basis for service-oriented computing. Component-based applications are a composition of components and alternative component implementations are used to realise specific functions.
the user interface feature and the app logic are distinguishable. On the other hand, the UML variability model in the MUSIC notation (cf. Figure 3.8 b)) is closer to the actual implementation. Component types and component realisations can be distinguished but the variation points are only implicitly defined by the ComplexComposition and its component types.

The predominant use of feature models or UML-based variability models stresses the need for variability models understandable by machine and humans. The use of MDD in self-adaptive software development seems to be the obvious way to increase intelligibility and efficiency during the development of those systems. Model-free self-adaptation (e.g. some reinforcement learning algorithms) takes already only a niche and would not support understandability of complex software systems.

![Feature model for the HelloWorldApp concept.](Image)

![UML-based variability model in MUSIC for the HelloWorldApp application.](Image)

**Figure 3.8:** Modelling variability.
3.6 Types of Architectural Adaptation

There are several ways to achieve architectural adaptation in software systems which we will in the following. McKinley et al. [98] name parameter adaptation and compositional adaptation as the two general types to realise software adaptation. Parameter adaptation also known as parameterisation involves the modification of dependent programme variables that determine the system’s behaviour. It does not affect the software’s actual structure and no new algorithms, components, or services can be added to the system. In contrast, compositional adaptation may exchange or add software components and allows for dynamic software re-composition at run-time. Compositional adaptation implies component-based software engineering. Figure 3.8 b) gives an example for compositional adaptation. The realisation CompositeHelloWorld is composed of two other component types, UI_Type and AL_Type whereas UI_Type can be realised by two distinct realisations.

Regarding adaptation cost and impact, Salehie and Tahvildari [128] differentiate between weak and strong adaptation, whereas parameter adaptation is weak and compositional adaptation is considered to be strong [128]. Weak adaptations are computationally cheap and the involved predefined and static algorithms have limited impact on the software from a software engineering perspective. Strong adaptation requires more resources to perform adaptations, e.g. when replacing or adding new components. Furthermore, it may trigger other strong and weak adaptations in sub processes. These considerations are particularly important when thinking about adaptation on resource-constrained devices. Strong adaptations have potentially more impact on the user, especially when components implement user interface functionality. Removing or exchanging user interfaces and their underlying functionality can cause serious distraction to the user. For instance, the small example from Figure 3.8 b) has two component realising user interface functionality that are exchanged depending on the context. On the other hand, parameter adaptation within user interface components may change the font size or the dialogues contrast, background colour, etc. These are typical low impact adjustments known from adaptive user interfaces (cf. Section 2.4.2) that are less dynamic than context-aware UC systems.

MUSIC [187] extends the compositional adaptation scheme by deployment adaptation and adaptation to external services in SOC environments. Deployment adaptation describes compositional adaptation over multiple devices. Components of a single application are deployed on different devices and multiple devices can provide components of the same type. The best fitting component composition among devices is selected. An often used example is the delegation of a user interface to better suiting device, e.g. from a smart phone to a smart TV. The adaptation to external services and the provision of services at the same time can implement a SOA. It is a generalisation of the deployment adaptation scheme and allows components to be provided as service instances. Reversely, an implementation of a component has not necessarily to be local. The system can adapt by integrating external services, e.g. web services as shown by Geihs et al. [50].

Adaptation can also be done over crosscutting concerns by employing adaptation by aspect weaving [172]. Crosscutting concerns contain requirements that cannot be encapsulated into individual modules because they are constantly needed within other modules. Examples are logging mechanisms or data encryption routines. Aspect-oriented Programming (AOP) extracts the crosscutting concerns into so-called aspects and weaves
the aspects at run-time at so-called pointcuts back into the components. Crosscutting concerns limit modularisation and hence variability. Without modularisation into distinct components, compositional adaptation cannot be applied. Alia et al. [172] present a solution for adaptation with crosscutting concerns by using AOP. As a basis they use the MUSIC compositional adaptation approach and provide extensions to the modelling notation and MDD processing chain. Adaptation by aspect weaving is a reasonable approach to integrate crosscutting concerns, but from a software engineering perspective it still has side-effects and only a couple of programming language support the interweaving of aspects. For this work we will rely on parameter adaptation and compositional adaptation with its sub types deployment and service adaptation. These types have been proved sufficiently for a variety of applications [187].

### 3.7 Uncertainty in Self-Adaptive Software

In Chapter 1 we motivated the need for user participation in SAS with a high degree of uncertainty. Uncertainty is a known problem in all autonomous acting computing systems [27, 28, 48]. It hinders or avoids the determination of the correct adaptation decision at run-time.

Generally, uncertainty can be distinguished between external and internal uncertainty [39]. External uncertainty is originated in the execution environment of the SAS. The SAS has no influence on this type of uncertainty but may be affected by it, e. g. if the decisions of the SAS rely on the current weather conditions. Internal uncertainty refers to the difficulty of determining the impact of adaptation. For example, Cheng and Garlan [28] see uncertainty directly in the MAPE-K adaptation loop. First, when the system has to detect the necessity for adaptation. Especially when working with sensor values, the algorithm should use a proper property and resource prediction to not trigger adaptation on false events (cf. Poladian et al. [117]). And second, when it has to detect an appropriate action. Compared to humans, the planning engine can base its decision only on a limited and fixed set of indicators. And third, there is uncertainty the action will be executed, i. e. the system needs to measure the effectiveness of the action and whether it was successful or not.

Adopted from Esfahani and Malek [40] we provide the following nine more general sources regarding the internal and external uncertainty:

**Uncertainty due to simplifying assumptions.** The SAS requires a perfectly tuned analytical model. The model is responsible for deriving the correct behaviour. Often, the assumptions of the underlying model do not hold during run-time. It may further be difficult for developers to make the correct assumptions and consequently, the resulting model is too simple.

**Uncertainty due to model/concept drift.** Especially white-box approaches, i. e. architectural adaptation makes simplifying assumptions or presume certain properties of the internal model of the system which will not hold in practice [186]. The machine learning community coined the term concept drift [166] to describe run-time changes that were not accounted for during the system's specification. The longer a system is in operative use, the higher the risk for concept drifts.

50  Self-Adaptive Software
Uncertainty due to noise. The analysis and planning components of the SAS rely on correct information provided by the monitoring component. However, the system has to deal with fluctuating, incorrect, or missing sensor values. The system should not trigger adaptation on every value change.

Uncertainty of parameters in future operation. There might be changes in the monitored phenomenon, i.e. the values to be measured can change over time. Not the current value of the sensor is important but rather the future values which need to be predicted based on past values. This prediction induces uncertainty.

Uncertainty due to human in the loop. The paradigm shift from data processing systems to ubiquitous computing systems leads to increased human involvement in the operation of self-adaptive software. A UC system depends on correct human behaviour but this is inherently uncertain.

Uncertainty in the objectives. Depending on the stakeholders' requirements, a SAS has one or more objectives to fulfil. The expression and specification of requirements is especially difficult for SAS. Different stakeholders have different requirements and UCS have their focus on the user's individual preferences. Developing self-adaptive software for multiple different users is a challenging task.

Uncertainty due to decentralisation. Ubiquitous computing involves many different systems that interact with each other. Adaptation can span over several of such sub-systems [187]. The overall knowledge can be distributed over many different sub-system, each of them having only limited influence on the behaviour of the total system. Developers of one sub-system do not necessarily have knowledge about the total system and the interplay between each sub-system.

Uncertainty in the context. A self-adaptive system, particularly when mobile, will be executed and used in different execution environment. Developers can hardly foresee all possible execution environments at design-time of the system. It is uncertain whether required resources are available in a particular environment or not.

Uncertainty in cyber-physical systems. Software systems and the physical world become increasingly intertwined and tightly integrated. The physical world brings uncertainty by nature and hence establishes non-determinism into the software system. However, the software can also affect the physical world resulting in uncertain conditions and behaviour.

Cheng et al. [26] focus on uncertainty in the specification of requirements, i.e. they address uncertainty in the objectives. They argue with incomplete information on the future execution environment of the SAS during development. The requirements for the system's behaviour may need to change at run-time in response to the changing environment. The requirements may have to include different degrees of uncertainty or some may be specified as incomplete. Therefore, Cheng et al. claim a new way of requirements specification that copes with incomplete information on the environment and the resulting incomplete behaviour about the behaviour of the system. Also, the evolution of requirements at run-time has to be supported. However, it must be guaranteed that critical high-level goals of the system are always met. Consequently, Cheng et al. ask for a new requirements specification language. The specific language elements need to be mapped to the actual architecture so that uncertainty is considered.
During architectural adaptation. With RELAX [165], Whittle et al. present a language to address uncertainty in SAS. They designed a vocabulary to enable analysts to identify requirements that may be relaxed at run-time when the environment changes. The RELAX system can then handle those changes.

Khan [193] and the MUSIC approach [187] address uncertainty during the development process by allowing for unanticipated adaptation. Unanticipated adaptation means that new functionality and algorithms can be added or removed at run-time by adding or removing software components. For example, new components can be provided by external service instances. However, this approach is very limited regarding spontaneous ad-hoc changes of the system's behaviour.

As presented above, Esfahani and Malek consider the human in the loop to be a great factor of external uncertainty [40]. Hence, it may seem ironically that we claim for more human involvement in the adaptation process. However, our aim is not to reduce internal or external uncertainty of SAS but rather we will provide users with tools and methods to deal with the inherent uncertainty that SAS in ubiquitous computing environments imply.

### 3.8 The Role of the User in Self-Adaptive Software

There is a high variety of application domains in self-adaptive software research. Traditionally, SAS addresses non-functional QoS optimisation in data processing systems. Typical users are system administrators or system operators, i.e. people with significant domain knowledge and expertise in computing systems. However, the necessary influence of humans is low. Autonomic computing and self-adaptation aim at automation and being independent of humans and uncertainty introduced by humans. But, humans have to intervene when the system does not know what to do, when it makes wrong decisions, or when it is not intended to do something on its own. However, automated systems can induce automation surprise [130], i.e. the system behaves converse to the operators expectations. Moreover, automation realises easy to automate tasks and leaves the infrequent difficult tasks to the user. This hinders operators to gain fruitful regular experience on the system [127].

Since the increase in popularity of ubiquitous computing, adaptive systems are more and more present in everyday environments with ordinary end-users. Examples are smart homes, smart phones, intelligent vehicles, or smart wearables like watches or glasses. Moreover, autonomous (mobile) robots enter people's everyday life, e.g. in rescue and healthcare scenarios or service robotic in general. SAS experiences a paradigm shift from adapting non-functional requirements solely to functional adaptation [40].

Self-adaptive software wants to provide the best service in any situation by adjusting itself to the situation. In case of non-functional QoS optimisation the impact for the user is limited. Of course, reducing network throughput, screen brightness, or increasing latency may affect the user negatively, but when adapting actual functionality users might become distracted or disrupted from their current task. In the worst case, a user is not able to carry out his current task anymore. Little attention has been given to the frequency of adaptations. Khan put it aptly: "so, the question arises, how much adaptation activities can be inflicted upon the user" [193]. While response time is crucial
in domains like robotics or data processing, high-frequent changes can be annoying or even disturbing in more user-oriented domains. Maybe a user does not want to have adaptations at all for a period of time.

To get a feeling on the effects of functional adaptation we conducted a qualitative user analysis with eight potential users using discount usability engineering on the first version of the Meet-U case study [219]. We focused on the usability of Meet-U and its adaptations to identify the effects of adaptations on the usability of an application. During the development of Meet-U, we already noticed several usability limitations of the underlying MUSIC self-adaptation concept. Therefore, we were eager to know whether the usability limitations would be experienced by ordinary users, too. Further, we wanted to uncover other issues regarding self-adaptation and usability.

Besides general usability issues, the evaluation revealed problems regarding Meet-U's adaptations in particular. Participants were surprised about sudden and non-notified adaptations and user interface changes. Although they liked Meet-U's anticipatory behaviour, they complained about a lack of control. Participants also recommended making adaptive functionality manually accessible although the system would decide different.

In Chapter 1 we elaborated the requirements of ubiquitous computing systems including human-computer interaction, trust, and individualisation by users. Due to their focus on automation and autonomous operation, SAS has significant limitations regarding user interaction, trust, and usability. External and internal uncertainty work reinforcing.

In contrast to empirical studies (like our user analysis or the case study described in Chapter 12), Wenpin Jiao [161] made achievements to describe an adaptive system formally. Their key questions is How effective is adaptive software?, i.e. how well does the system satisfy the user's expectations through adjusting its behaviour or configuration to tackle the changes in the environment. The proposed mathematical model includes the degree of change in the environment, the degree of adjustment of the system, and the degree of satisfaction to meet the requirements of the system, i.e. the users' expectations. The approach by Wenpin Jiao measures the degree of satisfaction in relation to the system's requirements. It does not involve actual human satisfaction indicated by the users. It is unlikely to express multifarious human experience in such a formal way. Therefore, empirical studies are mandatory when it comes to human-computer interaction.

### 3.9 Summary

Self-adaptation was conceived to automate processes and to build intelligent systems. The goal was to remove the human from the loop as much as possible. This works well for data processing systems which mostly employ weak parameter adaptation. Due to the paradigm shift, functional and strong compositional adaptation patterns become more

---

2Discount usability engineering [31] states that five participants are enough to detect as many usability problems as it could be found when using many more test participants [107]. A bigger number of participants lowers the benefit-cost ratio in qualitative user testing.

3The Meet-U case study will be described Chapter 12. There we will provide details on the target group, supported functionality, and the concrete realisation.
important. New application domains like smart homes, healthcare, smart phones etc. now also require intelligent, dynamic, and adaptive software. But in any case, SAS relies on a large amount of uncertainty which likely has a negative effect on potential users, whether they are administrator, operators, or end-users. However, the design on those systems does not foresee any user participation although the necessary mechanisms are partly available.

So far we used the term self-adaptive software which is rather general and our focus is on self-adaptive applications precisely. The term application implies a more end-user-oriented view on SAS. User participation is particularly relevant in interactive applications. This application-layer adaptation comprises software re-configuration for the entire application, i.e. adaptation is not limited to simple parameter adjustments, but could rather also include user interface adaptations.

In the following, we focus on white-box architectural adaptation using component-based software design. Component-based software is able to cover a broad variety of software realisations at great flexibility and is therefore perfectly suitable for ubiquitous computing applications.

Modelling variability with feature models, UML diagrams, or other proprietary techniques is inherently powerful. Hence, using a model-driven development process for the engineering of SAS is reasonable.

The actual input from control theory is limited, but it helps us to understand the motivation of the MAPE-K adaptation loop. We do not consider multiple control loops as this would imply an internal adaptation approach. External adaptation however, is conceptually cleaner, expandable, manageable, adjustable, and interchangeable. In combination with middleware-based adaptation we also achieve a separation of concerns between the adaptation logic and the application logic.

For further information on self-adaptive software we refer to the most relevant and recent surveys and studies in this field [72, 92, 128, 162, 163].
4 Self-Adaptive Applications – A Literature Study

From the perspective of autonomic computing and self-adaptive research, the impact on the user imposed by self-adaptive activities can be seen as a tradeoff [163]. Tradeoffs have a negative impact on the goals of self-adaptive software. There are only few works known from self-adaptive research that address user participation and other tradeoffs for the user imposed by self-adaptive software (see Chapter 5 on related work). We have to ask why user participation is practically not a major requirement in the development of such systems. In traditional software engineering, users are part of the stakeholder group as well as experts that apply for user goals [140].

Our hypothesis is that self-adaptation in domains like ubiquitous and mobile computing have a significant impact on the user and hence require a significant amount of user participation. Therefore, we conducted a literature study to understand the impact of self-adaptive software on users. Our goal is to show that many self-adaptation mechanisms, frameworks, or middlewares are evaluated with applications that would actually require a consideration of user requirements, but the presented mechanisms do not satisfy those requirements. Such user requirements can be intelligibility, trust, or usability in general as opposed to technical requirements like efficiency or performance. Before we go into our own literature study, we summarise the results from another study with a more general focus on the self-adaptation research community.

Weyns et al. did some outstanding work by being the first that elaborated the claims, tradeoffs, and supporting evidence for architectural self-adaptation. In two recent publications they present the results of an extensive literature study [162, 163]. Our hypothesis is supported by their latest findings in an analysis over a total of 121 publications [162]. They found out that the majority of researchers focus on a single concern only, like efficiency (55 %), reliability (41 %), or flexibility (28 %). For example, usability (6 %) plays only a minor role. Other human-related concerns like trust were not considered in the study. In case researchers considered multiple concerns in their work, they only focused on the positive effects of adaptation. 80 % of the analysed research work did not consider tradeoffs like user impact at all. Those that recognised tradeoffs postpone it to future work. According to Weyns and Ahmad, “most researchers on self-adaptive systems claim improvements of software qualities […]” but at the same time they “ […] simply ignore implications of self-adaptation”.

Coming along with the negligence of implications, the questions is how researchers got evidence for their claims. When we developed our first prototype of the Meet-U application with the MUSIC methodology and framework, we wondered why the profound implications for the user were not already considered during the MUSIC development. Referring to the results from Weyns and Ahmad, most of the research is
assessed using simple toy examples\(^1\) with only a minimal degree of evidence (95.8\%). Only very few (weak) empirical studies exist (2.5\%) and industrial applications of architecture-based self-adaptation were not existent at all. According to the authors “[…] research in architecture-based self-adaptation is still more exploratory than exploitative”.

The study by Weyns et al. had a general bias and did not focus on the examples used to motivate self-adaptive solutions. Moreover, they did not look at other tradeoffs beyond usability. In the remainder of this section we analyse existing self-adaptive software and applications in regard to user participation and non-functional user requirements like usability, trust, or intelligibility. We particularly pursue the following questions:

**Q1:** Which application domains are mentioned in publications?

**Q2:** What types of applications are used by researchers to evaluate their mechanisms?

Before answering question Q1, we first have a look at other studies and surveys and their view on application domains. While the survey done by Macías-Escrivá et al. [92] provide an extensive list of possible domains, Kakousis et al. [72] see the main application in mobile and ubiquitous computing, and Salehie and Tahvildari do not name any concrete application domains. Weyns and Ahmad [162] also analysed the application domains from the 121 studied publications. Quite surprisingly, only about 69\% of the works do consider an explicit application domain. However, they only name abstract domains which were mainly embedded systems (46\%) and web applications for e-commerce and information systems (30\%). The remainder is applied to robotics, multimedia, games, and transportation. The authors predict a growing interest in dynamic service composition domains, too. Due to lack of more concrete applications domains, we cannot precisely say whether these domains actually involve users and to what degree, although it is unlikely that there are no connections to administrators, operators, or end-users.

In the following two subsections we present results from a small and limited literature analysis to answer question Q1 and Q2. We selected publications from the beginnings in self-adaptation research at the end of the 1990s until today. Relevant work had to comply with the following criteria:

- The publication has to be clearly addressed to the self-adaptive community\(^2\).
- The self-adaptive software system employs an architectural adaptation approach as this is the predominant solution which we address, too.
- The provided solutions must be middleware-based, a framework, or a general systematic adaptation approach. Due to their complexity, we expect a more thorough motivation and evaluation by these works.
- The solution is used to manage/run adaptive software or applications.
- The publication has to include the full approach to achieve self-adaptation.
- If multiple publications for the same solution were found, only the latest is chosen.

\(^1\)Toy examples are not evaluated in their actual context of use but rather used to demonstrate a technical solution as simple as possible. Software engineering or socio-technical systems engineering require multiple comprehensive evaluations of a working prototype to understand impact and to identify problems [214, 140].

\(^2\)Sometimes context-aware software behaves very similar to self-adaptive software. Especially very early developments are not indicated as self-adaptive. We included such work whenever there was a clear evidence of an architectural adaptation approach.
We found a total of 41 publications including 38 example applications meeting the above criteria. Some of the publications contained multiple examples; some did not have any examples, ending up in a total set of 48 records. We did a manual search via Google and Google Scholar\(^3\) and used citations as well. Although the list of studied publications is not exhaustive, it however includes many major publications in the field of self-adaptive software. The complete list can be found in Appendix B.

For each publication, we determined the research domain, the application domain, and the type of application used in the evaluation, if any. For the application we further determined the maturity of the used prototype and the relevance of human factors, e.g. whether it was an interactive or server-based software in the background of human focus. Also, the type of evaluation has been determined.

### 4.1 Research and Application Domains

In our literature study we differentiate between the motivated research domain (meta domain) and the actual application domain of the provided examples. Although self-adaptive software is a research domain by its own, it needs to be put into a context. We assume that a record in our data set includes at least one motivation domain. In most cases, the application domain is more specific, given by the example application used.

Figure 4.1 presents both research and application domains. To address question Q1 we used all 48 data records for the analysis because publications with multiple examples had different motivation domains and publications without any examples also mentioned motivation domains.

The results show that mobile computing is the most widely used motivator for self-adaptive software (37.5\%), followed by service-based system (19\%), and ubiquitous computing in general (17\%). This is not surprising as all domains are highly dynamic, particularly mobile computing demands for self-adaptive software. Robotics is only used in 12.5\% of the publications even though it is perfectly suited. Although many of the self-adaptive concepts are also (implicitly) present in the robotics domains, we assume that the reverse does not hold – researchers from robotics do not publish within the self-adaptive software community. Remarkably, 10\% of the publications do not use a motivation domain at all. However, these are fewer publications without an explicit domain than Weyns and Ahmad found in their study. We assume this is due to our explicit focus on complete and distinct solutions for architectural self-adaptation.

For the application domain we analysed the 38 data records containing actual examples for the presented adaptation approach. The results for the application domains are summarised in Figure 4.1 b). The majority of solutions focus on the media domain (18.5\%), followed by healthcare, travel, and smart homes (each with 10.5\%). Information systems, social applications, and technician support software reached 8\%. 10.5\% provide an example application, but they do not specify a specific application domain. This is the case for robotic applications where researchers used a generic robotic scenario to demonstrate their approach.

\(^3\)Google Scholar is an Internet meta search engine for scientific publications: http://scholar.google.com/ (visited on 26/03/2014).
4.2 Example Applications

To answer question Q2 we have to look at the concrete examples used in the publications. We use three metrics to assess the examples: maturity of example, impact on users, and quality of evaluation. To the best of our knowledge, nobody has done this before.

The maturity determines how advanced the example is. This starts with simple textual descriptions and ends with a software system already in productive use. For the latter,
Weyns and Ahmad use the term *industrial application*. From the 48 data records only 79% contributed an example. Most of the examples were prototypes (54%) or only textual descriptions (21%). Industrial applications were not presented (cf. Figure 4.2).

![Figure 4.2: Maturity of provided examples.](image)

The impact on the user is classified into none, low, medium, and high impact. High impact is for applications in which adaptations cause a direct effect to the user, i.e. are functional adaptations. Medium is for applications that use a mixture of functional and non-functional QoS adaptation. Systems using solely QoS optimisations in the background are classified into the low category whenever they are indented to be used in

![Figure 4.3: Expected impact on users.](image)
a user-oriented domain. No impact can only happen in domains where no user interaction is expected and all adaptations are non-functional. Due to the lack of proper evaluations and descriptions, we had to guess the potential impact on the user from the provided information. Many of the examples were situated in the mobile computing domain applying adaptation to mobile software, i.e. scenarios in which user operate mobile devices. As a result, the majority of examples have been classified to have a high impact on user (71%). Assumed to have no impact on users were only 11% (cf. Figure 4.3).

The quality of evaluation is assessed to measure how well researchers evaluated their approach and to understand whether they considered tradeoffs of adaptation, too. Here we used the same terms as Weyns and Ahmad. The majority of publications (45%) included evaluations on the claims of adaptation, i.e. performance, re-configurability, availability, precision, etc. 39% did no evaluation at all and some others made only a proof of concept study without a quantitative evaluation (11%). Only 5% evaluated their work in a real world scenario to understand the impact and acceptance of adaptive applications (cf. Figure 4.4).

![Figure 4.4: Assessed quality of evaluations.](image)

In the following, we describe some example applications in more detail. For example, in the MADAM project, one of the requirements for the self-adaptation middleware was the support of user interface delegations to another device realised by deployment adaptation [189]. However, the delegation of a user interface to other devices without communicating this to the user may impose serious irritations. Within the MUSIC project, five trial applications have been developed [42, 187, 126]. All applications are from the mobile computing domain and help people in their everyday activities, e.g. support technicians, help (handicapped) travellers in the Paris Metro, support when First Aid is required, or help people to share content in social communities. All applications involve large amounts of user interaction and adaptations have a significant impact on the usability and trust. Khan [193] uses only a descriptive scenario to motivate his work. Although including many user interactions in the scenario description, the scenario is not evaluated regarding impact on the users.
The SEAMS\(^4\) community is one of the major research communities in the field of self-adaptive software. The SEAMS wiki\(^5\) names two example applications for self-adaptive applications: (1) a web-server-based news system called Znn.com, and (2) an automated traffic routing problem for an autonomous vehicle scenario. Although the goal of the Znn.com example is to successfully provide news at peak times with a high user load and the least interruption, it cannot neglect the customer's wishes and requirements regarding the contents. Autonomous vehicle scenarios are obviously scenarios with user involvement; whether there are autonomous vehicles carrying humans or completely autonomous drones interacting with other human road users. The SEAMS examples are also used by some of the publications, e.g. RAINBOW [188] or Stitch [207]. We see that even a research community with a strong bias on self-adaptation uses example applications which would need special structural concepts for user participation.

4.3 Summary

None of the analysed publications considered tradeoffs like usability, trust, or intelligibility. Only two did an evaluation in a real world scenario with real users. However, those evaluations rather focused on the general acceptance than on analysing tradeoffs explicitly. Some of the researchers evaluated their frameworks with developers to assess they suitability for developing applications. We did not include this type of evaluation in the study because we are interested in self-adaptive application usage.

None of the publications considered users in their approaches for architectural adaptation albeit the major motivation domains are mobile and ubiquitous computing – both are domains which likely include users. Favourite example applications are adaptive video players or conferencing applications. Both types include a high degree of user interaction and have a significant impact due to autonomous adaptations. None of the provided examples are real world applications that are already in productive use. Considering that the argumentation for self-adaptive and autonomic computing is in favour of automation and user should be removed from the loop as much as possible, the majority of research works address application domains that probably require users within the loop.

This literature study and the results from Weyns et al. [162, 163] reveal a dilemma of the research field: on the one side there is a variety of comprehensive and sophisticated approaches to achieve architectural adaptation on the technical layer and on the other side nobody has ever validated if their approaches are reconcilable with users' expectations.

Although the study showed no real evidence for a lack of user participation (due to missing evaluations), our own usability study (cf. Section 3.8), experience during development, and automation theory (cf. Chapter 1) make user consideration necessary.

\(^4\)SEAMS stands for Software Engineering for Adaptive and Self-Managing Systems.

5 Related Work

The field of self-adaptive software is multifaceted and also multidisciplinary. Chapter 1 already outlined general related work for the interdisciplinary fields of usability, interaction design, and trust. In this chapter we present developments from the (self-) adaptive community that address at least one of the following issues: intelligibility, trust, usability, end-user programming, user participation, fuzzy-based adaptation, user activity detection, user in the loop, or adaptive user interfaces. To the best of our knowledge there is no comparable approach to (systematically) integrate the user in an adaptive system. We assume this is firstly due to the lack of experience with real applications so that nobody could encounter any human-related problems with self-adaptive applications. And secondly, self-adaptive and autonomic computing research strives for the removal of users from the decision loop to avoid human failures.

5.1 User Participation

Users can already participate in software activities and their design in a few ways. Some approaches allow a modification of the run-time model while others enforce end-user design and development of applications so users can tailor software to their needs.

End-user development of component-based software is proposed by Mørch et al. [100]. Although the provided high-level component-based view allows abstract software compositions, it still demands for advanced knowledge on software architectures. Moreover, the presented approach does not consider adaptive applications and is focused on ordinary desktop software. It further lacks of a proper evaluation with the intended target group.

There have been several attempts within context-aware computing to provide users with the possibility to design dynamic and adaptive software on their own. A well-known example is a CAPpella [34] from Dey et al., which is stream-based and every context sensor is represented as a stream. From the graphical user interface, the user can select the required streams and define beginning and end times for recording. The user captures the context-aware behaviour in-situ and a CAPpella is trained on this data. The provided examples are simple, but the general approach is quite user-oriented and users can easily define their own context-aware behaviour. However, a CAPpella does not provide any systematic adaptation approach.

Sousa [141] proposes new concepts for smart spaces like design meshing and pliable apps. With design meshing users can individually develop their own components and artefacts in a ubiquitous multi-user environment. Different users can then combine their artefacts into a mesh. Pliable apps address the mobile computing domain and allow users to design structural adaptation and to change the behaviour at run-time. However,
the author proposes modelling concepts which are still quite technical and users need to understand the concept of component-based software engineering. The description of run-time behaviour is limited to context event triggers. The authors are further not very detailed on the way run-time changes happen, but users can also make changes at the run-time structure. Example applications for the concepts of design meshing and pliable apps are not provided. Both concepts employ the TeC framework [142] for end-user design, deployment, and evolution of applications for smart spaces. This is what Sousa calls user-controlled: “user-controlled is the fact that adaptation is described at the design level using machine-interpretable constructs.” We think it is not possible for non-expert end-users to model systems, connect components, or even program components with machine-interpretable constructs. Although the authors argue that TeC has a higher level of granularity than programming constructs, it still can only be used by experts. However, with appropriate user interfaces that abstract sufficiently, users might be able to compose their own smart apps in ubiquitous computing environments. Our proposed solution will not cover end-user development in terms of application design and component creation but rather focus on the interaction between a user and the self-adaptive software and adjustment to the user’s needs.

In our terms, end-user design, development, and deployment correspond to the structural participation dimension (cf. Chapter 1). In this work we only focus on the temporal and behavioural dimensions which are not addressed by any of the presented works.

Adjustable autonomy is a term introduced in the field of autonomous systems [35]. The systems were mainly used in space mission and aircraft control. Adjustable autonomy should overcome the automation paradox [11] and automation surprise [130]. The idea is to adjust the level of autonomy between manual operation and autonomous operation. Later, this term was transferred to agent-based systems [20] and context-aware systems [53]. Especially in the field of agent-based systems the term adaptive autonomy has been established [12]. Hardian [53] proposes a middleware solution to balance autonomy and user control in context-aware systems. However, the main suggestion of Hardian is to reveal internal information to the user, i.e. context information, preferences, and adaptation logic. Unfortunately, the author did not provide any more details on how this will be realised and he does not provide the actual mechanism to balance control. In our opinion, revealing such information may increase transparency and control, but it can also overwhelm the user with unwanted and unnecessary information. Moreover, small user interfaces on smart phones, tablet computers, or other ubiquitous devices do not allow for a reasonable interface design with much information.

In Chapter 2, we already introduced automation applications for mobile phones (e.g. Smartactions, Tasker, or Automagic). These applications have only simple rules that trigger certain phone-specific events. It is not possible to use these applications to control other applications in turn. However, due to their focus on mobile devices they have well-engineered user interfaces that allow for displaying complex relationships of rules, triggers, and events.

5.2 User Activity Detection

Mobile devices especially smartphones are equipped with a variety of sensors. The sensors can be used to recognise the current activity of users. We will use activity detection to
classify the current user’s interaction activity with the device. Unlike activity detection, algorithms for activity recognition are subject of several research projects [85, 102]. For example, Lau [85] uses smartphone sensors to detect and classify different types of gestures and movements. Mühlbrock et al. [102] performed a similar approach for activity detection in an office environment, although they used multiple standalone sensors instead of a smartphone.

Falaki et al. [41] developed a monitoring system called SystemSens to capture usage context of smartphones. It runs as a background service and collects data on battery, network connectivity, CPU usage, phone calls, or application activity. Primarily, system Sense is a logging tool for debugging during application development. However, the presented system is a pure logging system and does not perform any classification on the measured data.

Shye et al. [136] did a sixth-month study with 25 users to understand real user activity on smartphones. Their focus was on typical user activity and the energy consumption over time. They found CPU and display activity to be the reason for high energy consumption. Hence, CPU and display activity can be used to detect user activity. There is also a large variation in usage patterns across the participants. This supports our assumptions of a high variance of typical users. Self-adaptive applications have to address this variance. However, Shye et al. did not try to classify the interaction activity more specifically. There are several similar approaches [47, 59], for example Froehlich et al. [47] who developed the MyExperience system to collect sensor data and real in-situ usage data from users.

In contrast to the presented approaches, we are not interested in actual usage or activity recognition, but we rather like to know to what degree the user is currently interacting with the device. To the best of our knowledge there is no previous work on such interaction activity detection and classification.

### 5.3 Fuzzy-Logic-Based Self-Adaptation

Fuzzy logic has been applied to many domains in science and engineering. In this section we will focus on the fuzzy-logic-based reasoners for (self-) adaptive software and mobile computing. Four of six presented works were published within the last year. There is a clear demand for uncertain reasoning within self-adaptive software.

RELAX [165] is a requirements language for self-adaptive systems that addresses uncertainty in terms of fuzzy logic. The semantics of RELAX are defined in fuzzy branching temporal logic (FBTL). Each expression in RELAX has its corresponding pendent in FBTL. However, it is limited to the requirements definition of adaptive systems and the work by Whittle et al. does not show how the fuzzy-defined requirements are mapped to an architectural adaptation approach.

Chuang and Chan [184] developed a fuzzy-based adaptation middleware for QoS optimisation. Unlike Whittle et al., they present a full adaptation model. They use a hierarchical model for QoS metrics to abstract from concrete sensor values. One or more sensor values define one QoS metric via fuzzy inference. Beside these contextual QoS metrics, the authors define user satisfaction QoS metrics. These are fuzzy variables that fuzzify user-oriented QoS parameters like media smoothness. The adaptation inference
engine will then use the user satisfaction and contextual metrics to deduce required adaptation decisions. In this work we also use a hierarchical model to deduce higher-level fuzzy metrics from sensor values. But unlike the QoS-oriented approach by Chuang and Chan, we incorporate this in MAPE-K-based functional adaptation. A similar concept for fuzzy QoS optimisation is followed by Beggas et al. [176], which we will describe further down.

Provensi et al. [121] use fuzzy profiles to improve context interpretation. They integrate the three-step fuzzy reasoning approach in the MAPE-K adaptation loop in the components analysis and planning to deal with contextual imprecision. Fuzzification and inference is done in (context data) analysis while defuzzification is done in planning. A video streaming scenario was used to demonstrate the adaptive adjustment of video quality according to available bandwidth. The actual adaptation decision is not made with the fuzzy approach. The authors do not provide information where fuzzification, defuzzification, and rules are specified. We follow a similar approach but unlike Provensi et al., we separate all fuzzy-related context reasoning into the analysis component.

Yang et al. [168] present the SFSA framework for self-adaptive mission-critical software. SFSA employs component-based software and aspect weaving for adaptation. Sensor values can either directly influence adaptation or fed into a fuzzy process first. Unlike the other presented works, Yang et al. include a fuzzy rule editing and testing interface. However, their focus is on embedded server systems with little user interaction.

Parallel to this work, Beggas et al. [176] integrated fuzzy control into the adaptation reasoning process of the MUSIC middleware. Like Chuang and Chan, they focus on non-functional QoS optimisation and enhanced the planning with fuzzy reasoning capabilities. The utility of an application variant is directly determined by the fuzzy reasoning process, i.e. each rule has only one consequence, the utility variable. This approach is straightforward, but has two drawbacks: first, with many input variables, the resulting utility for the different variants is difficult to foresee in advance. Second, the person who specifies the fuzzy rules needs knowledge on the internal application structure, the QoS parameters, and the available variants. This is a cumbersome task even for experienced expert users. Hence, we propose a hybrid approach that lets users define high-level fuzzy metrics based on predefined context information. These metrics are hard-wired by the application developer and allow a better prediction of the actual behaviour.

### 5.4 Learning-Based Self-Adaptation

While machine learning is widely used in domains like robotics, there are not many examples of learning-based frameworks for self-adaptive software. For instance, Kim and Park [79] employ a self-adaptive reinforcement learning algorithm for online planning of robot behaviour. A robot is adjusting its behaviour by learning from prior experience. Adaptation plans are changed as response to environmental changes.

Tesauro [147] propose a hybrid approach that combines analytical modelling with reinforcement learning. The approach assumes that a model for training the algorithm exists. However, they focus on the problem of resource allocations to applications in data centres and real-time systems in general. Early on, Tesauro realised that the knowledge component of the MAPE-K loop needs periodic redesign to cope with
evolving requirements. The authors also noted that a MAPE-K planning component might require more generative planning than reactive reinforcement learning planning. Reinforcement learning requires states and actions to be defined which we do not have described explicitly in our adaptation approach. Agents using reinforcement learning use environmental exploration to receive positive or negative reward depending on the outcome. In combination with users, wrong decisions have a stronger effect than correct decisions.

Unlike the previous works, FUSION [186] claims to provide a general-purpose approach for self-adaptive software. Esfahani et al. provide a black-box approach for engineering SAS, i.e. adaptation decisions are made using abstractions that do not require any knowledge on the software's internal structure. They use a feature-based adaptation whereas each feature describes a capability of the system. A learning algorithm induces functions from observed metrics. The FUSION framework is not restricted to a particular learning algorithm. For their evaluation, Esfahani et al. used a M5 model tree, a special type of decision tree. Although the FUSION black-box approach is comprehensive we cannot apply it to our white-box architectural adaptation concept. Moreover, FUSION cannot deal with uncertainty introduced by sensors, it relies on accurate values. It further needs sufficient training data to perform precise learning. As a white-box it does not specify the required mapping from features to software component and the actual adaptation mechanisms. Additional work would be required to apply fusion to different application domains.
6 Fuzzy Control

The term fuzzy control describes control systems that are based on fuzzy logic. Essential parts of fuzzy logic are so-called fuzzy sets. Zadeh proposed the concept of fuzzy sets in 1965 [171]. In 1996 he published fuzzy logic [170] although others did before him. Fuzzy theory has been widely and successfully used in control systems [87] whenever reasoning with vague information is required. There are several recent examples, in which fuzzy theory has been applied to self-adaptive software [13, 176, 16, 121, 151, 168]. We discuss some of those works in Chapter 5 on related work.

The term fuzzy refers to the fact that the underlying logic can deal with uncertain variables. In fuzzy logic, a statement can be either true or false and at the same time it can be neither true nor false. Instead, logical variables can take real values between 0 and 1. This is different to classical logic which can only assign values that are either true or false.

A controller based on fuzzy logic takes precise physical (discrete and continuous) input values and determines their fuzzy membership according to predefined linguistic terms and evaluates them by applying IF-THEN rules. The resulting, still fuzzy, values are then converted back to precise output values. These three steps are called fuzzification, inference, and defuzzification (cf. Figure 6.1).

Humans are able to control process systems with imprecise, but directed information in a way that it leads to an optimal result. For example, someone would set a room heater thermostat higher when the perceived temperature is cold. Classical logic systems are either undecidable (e.g. first-order logic) and/or cannot properly deal with uncertainty (two-valued logic). However, fuzzy systems come at the cost of partially uncertain results. Fuzzy logic is neither absolutely exact nor inexact, it can only express a membership to a certain degree of exact or inexact. On the other hand, fuzzy control has several advantages over classical control systems:

1. Complex problems can be described with fuzzy rules in natural language.
2. The solution to the problem can be cast in terms that administrators, operators, or even end-users can understand.
3. Natural language describes uncertainty and vagueness with linguistic terms and simple IF-THEN rules.
4. The rule base can be easily modified and tweaked to improve or significantly alter system performance.
5. Conflicting rules do not compromise the control process.
6. There is no need for precise and noise-free inputs.
7. There is no need for a precise mathematical control model.
Due to the above reasons, fuzzy logic is well suited for self-adaptive systems with humans in the loop. Yang et al. showed that fuzzy control is more trusted by users as fuzzy logic follows human logic [169]. Therefore, we will use fuzzy logic for two user participation concepts in Chapter 11. The first concept uses fuzzy logic to define user-oriented adaptation policies, i.e. users can use natural language terms to define robust adaptive behaviour without bothering with internal reasoning technicalities. The second concept uses fuzzy logic to let users define situations that feed into a learning process to classify application variants according to the situations.

The rest of this chapter will introduce the definitions and theoretical foundations of fuzzy control systems.

6.1 Fuzzy Sets

Fuzzy sets constitute the foundations for fuzzy logic. Fuzzy set theory is different from conventional set theory. Each element of a given set belongs to the set in some degree. Conventional set theory requires for each element to be fully included or completely excluded from the set. The degree to which an element is included in a given set is called the grade of membership within the interval of 0 and 1. It is important to understand that the membership value does not denote any kind of probability.

A fuzzy set $A$ is defined as

$$A := \{ (u, \mu_A(u)); u \in U; \mu_A(u) \in [0, 1] \}. \quad (6.1)$$

Whereas the universe $U$ denotes the universal set with $u$ being an element of this set. The universal set $U$ follows a function $\mu_A$ that maps to the closed interval $[0, 1]$: $\mu_A : U \rightarrow [0, 1]$. $\mu_A(u)$ specifies the degree of membership of an element $u \in U$ within the fuzzy set $A$. $\mu_A(u)$ is called membership function and the interval $[0, 1]$ is the membership space $M$. In case the function $\mu_A(u)$ maps onto the discrete values 0, 1 than $A$ is a conventional set. Hence, it can be concluded that fuzzy set theory generalises conventional set theory.

Two fuzzy sets $A$ and $B$ are equal ($A = B$) if

$$\mu_A(u) = \mu_B(u); \forall u \in U. \quad (6.2)$$
A fuzzy set $A$ is empty ($A = \emptyset$) if

$$\mu_A(u) = 0; \forall u \in U.$$ (6.3)

A fuzzy set $A$ is a subset of a fuzzy set $B$ ($A \subseteq B$) if

$$\mu_A(u) \leq \mu_B(u); \forall u \in U.$$ (6.4)

The complement $A'$ of fuzzy set $A$ is defined as

$$A' := \{(u, 1 - \mu_A(u)); u \in U; \mu_A(u) \in [0, 1]\}.$$ (6.5)

The union of a fuzzy set $A$ and a fuzzy set $B$ ($A \cup B$) within the same universe $U$ is again a fuzzy set $C$ whose membership function is defined as

$$\mu_C(u) := \max\{\mu_A(u), \mu_B(u)\}; \forall u \in U.$$ (6.6)

The intersection of a fuzzy set $A$ and a fuzzy set $B$ ($A \cap B$) within the same universe $U$ is again a fuzzy set $C$ whose membership function is defined as

$$\mu_C(u) := \min\{\mu_A(u), \mu_B(u)\}; \forall u \in U.$$ (6.7)

**Figure 6.2:** Common types of membership functions of a fuzzy variable $x$. Only singleton and piece-wise linear functions are defined by the IEC [65].
There are several common types of membership functions, particularly triangular and trapezoidal functions, to define the membership of a fuzzy variable. Figure 6.2 illustrates the most common types with the function-specific parameter set \( \{a, b, c, d\} \). However, the Fuzzy Control Language (FCL)\(^1\) defines only singleton and piece-wise-linear membership functions. In our implementation we use the FCL-compatible reasoner jFuzzylogic [29] that also supports other types of membership functions.

### 6.2 Fuzzy Logic

Fuzzy logic is a \( n \)-valued logic whereas classical logic is a two–valued logic denoting truth and falsity of a statement. For any given \( n \), the truth values are usually labelled by the rational numbers in a given interval. For fuzzy logic this interval, i.e. the membership space is \([0, 1]\).

Fuzzy logic employs fuzzy set theory as its basic tool to achieve reasoning with uncertainty. It uses so-called linguistic terms to describe fuzzy variables. Linguistic terms are expressed in a natural and hence imprecise way. Considering a fuzzy variable \( \theta \) that measures the current temperature in a room. The temperature value can be classified into the three linguistic terms low, medium, and high. Classical logic would need sharp interval borders to induce whether the current temperature is low, medium, or high. The fuzzy logic concept is closer to the actual human thinking, which is not bound to such precise intervals.

In fuzzy logic, each linguistic term is represented by its membership function associated to a fuzzy set. Thus, the borders between the different functions could be smooth and overlapping instead of sharp and separated. The individual types of membership functions relate to special (vague) linguistic expressions. For example, to map the linguistic term almost, a triangular function is well suited. For the term about we could use the Gaussian function and for approximately between a trapezoidal or shoulder function.

As in other logic systems, the variables are used to express facts and rules. Listing 6.1 provides an example for a fuzzy control system controlling the temperature in a room with a heater and a fan. All rules are equally important – the value of a fuzzy variable could be in two different fuzzy sets. A fuzzy reasoner will evaluate all rules to determine the membership of the fuzzy output variables (heating in this example). The THEN operator separates antecedents (left side) from consequences (right side).

```plaintext
Listing 6.1: Simple rule base used by a fuzzy controller to adjust the heating in a room.
1 Rule 1: IF temp IS normal THEN heating IS medium
2 Rule 2: IF temp IS cold THEN heating IS high
3 Rule 3: IF temp IS normal AND time IS evening THEN heating IS off
4 Rule 4: IF temp IS warm THEN heating IS off
5 Rule 5: IF time IS NOT day THEN heating IS low
6 Rule 6: IF time IS morning THEN heating IS high
```

\(^1\)The Fuzzy Control Language (FCL) is defined by the International Electrotechnical Commission (IEC) [65] undertaking the effort to standardise fuzzy control systems.
Basic fuzzy logic provides AND, OR, and NOT operators to combine multiple linguistic terms (AND and OR operator) or to negate the value of a linguistic term (NOT operator). All operators can be assigned to antecedents and consequences. Figure 6.3 shows the application of the three operators on linguistic terms using fuzzy set theory. However, apart from the basic \( \min \) and \( \max \) operations, other norms for AND and OR operators can be used in practice. Table 6.1 lists all methods supported by the FCL.
Compared to conventional control systems, context-aware and self-adaptive software has a large number of context dependencies and QoS parameters that may cause an explosion of the rule base. Hierarchical fuzzy systems \[167\] reduce the number of rules by hierarchically inferring fuzzy variables and grouping them into abstract variables.

### 6.3 Fuzzy Reasoning

Fuzzy reasoning involves the three steps fuzzification, inference, and defuzzification. Fuzzification is the assignment of a membership function to a linguistic term of a fuzzy variable. In other words, linguistic terms like low, medium, or high will be assigned to a sharp input variable (e.g. temperature) and also output variables like heating or fan. The linguistic terms have to cover the entire value range of the sensor or the actuator, i.e. every possible value of the fuzzy variable needs to be associated to at least one linguistic term. In practice, the membership functions for the terms overlap and values belong to multiple fuzzy sets. Each linguistic term can have a different type of a membership function, depending on the requirements (cf. Figure 6.2). Figure 6.4 shows a reasonable fuzzification for the three fuzzy variables temp, time, and heating.

![Figure 6.4: Exemplary fuzzification of the fuzzy variables temp, time, and heating.](image)
The actual reasoning can be done once the rule base is created and all contained variables are fuzzified according to the above instructions. This step is called inference. When the reasoning process starts, the current values for the sharp input variables are determined. For example, the temperature sensor measures $17^\circ C$. Once all values are determined, the rules are evaluated. Multiple rules can determine the same output variable. For example, Rule 1 and Rule 2 from Listing 6.1 can be both activated when the sharp sensor value $17^\circ C$ is associated to the linguistic terms cold and normal. Both rules determine the output variable temp and as all rules will be evaluated, we have to accumulate the results. Determining the maximum of all output fuzzy sets is the most common method (OR conjunction). Table 6.2 lists all methods supported by the FCL. Accumulation results in a single fuzzy set with a specific membership function for each output variable.

Once the final fuzzy set has been established, a precise output value needs to be determined to control the system. This step is called defuzzification. Several methods allow defuzzification from the final set (cf. Table 6.3). Calculating the centre of gravity (COG) is the standard method.

Now, we assume a current temperature $\theta = 16^\circ C$ and the current time at $t = 6 \text{am}$ to demonstrate fuzzy reasoning for the room heating example (see dashed lines in Figure 6.4). We first evaluate all rules from Listing 6.1 to the given input (inference):

1. Rule 1 is activated with a 0.2 membership for the normal set and results in $\mu_{\text{medium}}(H) = 0.2$.
2. Rule 2 results in a 0.4 membership for the cold set and results in $\mu_{\text{high}}(H) = 0.4$.
3. Rule 3 does not apply.
4. Rule 4 does not apply.
5. Rule 5 results in a 1.0 membership for the not day set and results in $\mu_{\text{low}}(H) = 1.0$.
6. Rule 6 results in a 1.0 membership for the morning set and results in $\mu_{\text{high}}(H) = 1.0$.

The two results for $\mu_{\text{high}}(H)$ are accumulated using the MAX method: $\mu_{\text{high}}'(H) = \max\{0.4, 1.0\} = 1.0$. Finally we defuzzify $\mu_{\text{high}}'(H)$ with a sharp value for heating of 51.41% using the centre of gravity (COG) method. Figure 6.5 illustrates the resulting fuzzy set after rule inference.

The complete FCL control file for this example is available in Appendix A.

![Figure 6.5: Accumulation and defuzzification result using the centre of gravity (COG) method after the evaluation of all rules from Listing 6.1 (temperature $\theta = 16^\circ C$, time $t = 6 \text{am}$).](image-url)
**Table 6.2:** Accumulation methods supported by the FCL to combine the results from several distinct fuzzy rule evaluations.

<table>
<thead>
<tr>
<th>Accumulation Method</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum (MAX)</td>
<td>$\max{\mu_A(u), \mu_B(u)}$</td>
</tr>
<tr>
<td>Bounded Sum (BSUM)</td>
<td>$\min{1, \mu_A(u) + \mu_B(u)}$</td>
</tr>
<tr>
<td>Normalised Sum (NSUM)</td>
<td>$\frac{\mu_A(u) + \mu_B(u)}{\max{1, \max{\mu_A(u), \mu_B(u)}}}$</td>
</tr>
</tbody>
</table>

**Table 6.3:** Defuzzification methods supported by the FCL.

<table>
<thead>
<tr>
<th>Defuzzification Method</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centre of Gravity (COG)</td>
<td>$X = \frac{\int_{\text{start}}^{\text{end}} x\mu(x) , dx}{\int_{\text{start}}^{\text{end}} \mu(x) , dx}$</td>
</tr>
<tr>
<td>COG for Singletons (COGS)</td>
<td>$X = \frac{\sum_{i=1}^{p}[x_i\mu_i]}{\sum_{i=1}^{p}[\mu_i]}$</td>
</tr>
<tr>
<td>Centre of Area (COA)</td>
<td>$X = x', \int_{\text{start}}^{x'} \mu(x) , dx = \int_{x'}^{\text{end}} \mu(x) , dx$</td>
</tr>
<tr>
<td>Left Most Maximum (LM)</td>
<td>$X = \min(x'), \mu(x') = \max(\mu(x)); x \in [\text{start}, \text{end}]$</td>
</tr>
<tr>
<td>Right Most Maximum (LM)</td>
<td>$X = \max(x'), \mu(x') = \max(\mu(x)); x \in [\text{start}, \text{end}]$</td>
</tr>
</tbody>
</table>

$X$ = result of defuzzification; $x$ = output variable; $p$ = number of singletons; $i$ = index; $\mu$ = accumulated membership function; $\text{start}$ = lower limit for defuzzification; $\text{end}$ = upper limit for defuzzification
Part II

Concept and Solution
7 User Participation – Overview

Our solution for user participation in self-adaptive software focuses on component-based software using an external middleware-based adaptation approach (cf. Chapter 3). For each of the provided concepts we will consider the specific properties of ubiquitous computing systems (cf. Chapter 2) to improve usability, trust, and unobtrusiveness in uncertain environments. We will address intelligibility and transparency of adaptations with our notification concept, but the focus is on the controllability of adaptations.

We identified two views on user participation, a user view and a system view. From a user’s perspective, participation can be distinguished between short-term and long-term participation:

**Short-term user participation.** Whenever the current application behaviour does not meet the user’s needs, he should be able to override or alter this current behaviour. These changes do not influence the application behaviour generally and in the future the adaptation engine will always decide in the same way as it would have prior the user changes. This can be seen as unplanned participation that is only implicitly done by the user. The user is not aware of actually modifying the behaviour, instead he interacts normally with the application. It is up to the participation concept to process the implicit user feedback appropriately. Short-term user participation addresses uncertainty due to sensor noise, uncertainty in the context, or uncertainty due to the human in loop.

**Long-term user participation.** When the application behaviour does not meet the user's requirements more than once, he might want to influence the behaviour in the long term and persistently. Long-term user participation is planned participation which is explicitly done by the user to modify the application. During modification of the

---

Figure 7.1: Integration of user participation into the MAPE-K adaptation loop.
application, the user is aware of the modification process. Compared to short-term participation, a user requires more knowledge and expertise on (self-)adaptive software. Long-term user participation addresses uncertainty due to simplifying assumptions, model drift, or uncertainty of parameters in future operation.

The integration of short-term and long-term participation into the MAPE-K adaptation loop is depicted in Figure 7.1. The participation management component interacts with the planning engine of the adaptation middleware. While for short-term participation this direct connection is sufficient, long-term modification further requires the modification of the knowledge base and the adaptation model. However, the planning engine is responsible in both cases. Notification and feedback mechanisms implement a bi-directional communication infrastructure between the adaptation middleware and the user (cf. Chapter 9).

From the system’s perspective, we identified three dimensions that affect user participation, namely moment of adaptation (temporal), application behaviour (behavioural), application structure (structural) (cf. Figure 7.2) [219]. These dimensions characterise the parameters for user participation. In each dimension, the degree of participation is somewhere between full user participation and no participation at all. This relates to the eight degrees of automation defined by Sheridan and Parasuraman [134]. They start from a system in which the system offers no assistance at all and go over to an autonomous system in which the user is completely ignored (cf. Chapter 2). A self-adaptive application belongs to the latter. Our proposed mechanisms for user participation will shift automation to a lower degree in any case. However, our goal is not to build applications that are completely controlled by the user. We focus on the harmonisation between application autonomy and user participation.

![Dimensions of user participation](image)

**Figure 7.2:** Dimensions of user participation: great variety between manual user participation and autonomous operation.
In the remainder of this chapter we provide an overview of the participation mechanisms in each dimension. We concentrate on the temporal and behavioural dimensions and leave the structural dimension for future work. From the user's perspective, one can say that short-term participation is addressed by the temporal dimension, whereas long-term participation matches to the structural and behavioural dimensions.

### 7.1 Moment of Adaptation

The temporal dimension determines the entity which controls the moment of adaptation, i.e. which entity triggers the adaptation process. In the MAPE-K adaptation loop, the system analyses information retrieved from sensors or reasoners and the planning engine decides which variant of the software will be selected for instantiation. The user might not always be comfortable with a certain moment of adaptation and its consequences if the system has made the decision on its own. Unlike the structural and behavioural dimensions, the temporal mechanisms address short-term participation of users.

**Manual selection of application variants.** Self-adaptation and autonomic computing do not foresee any participation on the adaptive application. All changes are performed by the self-adaptive software system itself. Ad-hoc changes due to unexpected incidents or changes in the routine are not possible. We will provide mechanisms to intercept the adaptation planning process so that users can select a different application variant than the currently active variant. Important questions are 1) how to override the system's decision and 2) how to resume the autonomous operation mode.

**Pre-controllability of adaptations.** Controllability of adaptations refers to the guidelines for usable interface design (cf. Section 2.5.1). Pre-controllability lets users decide what to do before an adaptation happens. A user might want to accept or reject the adaptation. When an adaptation is inappropriate for the current situation but not generally unacceptable, a user might want to postpone the adaptation. The system will then keep the current application variant, but will remind the user after a specified interval on the postponed adaptation via a suitable notification. The user can again decide whether to accept, reject, or postpone the adaptation. Notifications play a crucial part for controllable adaptations. From notifications users can decide what type of feedback to give to the system.

**Post-controllability of adaptations.** Instead of asking the user prior every adaptation (see pre-controllability), adaptations will be executed first. Subsequently, a user can undo, accept, or postpone the adaptation. When reverting an adaptation to the previous state, the challenges are the same as with the manual selection of application variants: because the user overrides the autonomous behaviour, the system must return to autonomous operation mode eventually. Postponing adaptation works the same way as with pre-controllability.

**Toggle the self-adaptation mechanism.** Beyond the previously described micro participation concepts, we also propose a mechanism to temporarily shut down the entire self-adaptive behaviour of an application. Once disabled, it is up to the user whether a default application variant should be instantiated or the current
running variant will be kept. A default application variant has to be specified by
the application developer in advance. Moreover, the user will able to suspend the
adaptive behaviour for a particular time span or persistently.

Consider the user's perceptual focus. Adaptations may have a negative impact on
the application's usability, for example users can be disturbed or distracted from the
current task. Usability is “the extent to which a product can be used by specified
users to achieve specified goals with effectiveness, efficiency and satisfaction in a
specified context of use” (cf. Section 2.5.1). Hence, the current context of use has
much relevance for the usability. This includes the current task and environmental
conditions. Hence, to improve usability of adaptations, we have to consider the
context of use. A user focus defines the perceptual focus on component-level,
i.e. a developer groups components into so-called user focuses that most likely will
be used in a specific context of use (cf. Section 10.5). For example, components
relevant for navigating the user will constitute one user focus group. The self-
adaptive software will not be adapted autonomously if the current application
variant contains a user focus group.

Consider the user’s interaction activity. As an extension to the user focus concept, we
also consider the user’s current interaction activity with the device. This allows a
more dynamic identification of the current context of use compared to predefined
static user focus groups. In Section 10.6, we present a solution that uses smartphone
sensors to detect the current interaction activity. With knowledge on the current
interaction activity, we can decide how to adapt appropriately. A user who is
currently entering a text within an application would not appreciate adaptations
until he is finished. Instead, we propose to use different types of notifications
depending on the current activity, e.g. unobtrusive notifications on high activity
and obtrusive notifications when the user does not pay attention to the device.

7.2 Application Behaviour

The behavioural dimension addresses the actual adaptation logic. At design-time,
developers specify the system’s behaviour by using an adaptation policy, e.g. utility
functions or actions. This approach can be too static for some applications, especially
when the user wants to configure the application behaviour according to his individual
preferences and needs (cf. Chapter 2). Modifying the behavioural dimension refers to
long-term user participation. A user plans to make changes in the behaviour explicitly
and not for a particular moment, but for every further execution. This is different to the
temporal dimension in which the user or the systems decides ad-hoc what should be done.
We now describe participation mechanisms belonging to the behavioural dimension.

Modification of the reasoning policy. In Section 3.4 we have seen different reasoning
policies. To modify the behaviour, the applied reasoning policy has to be
changed. Utility functions are powerful in combination with component-based
software engineering, but they are also error-prone and suffer from cumbersome
management. In Section 11.1.2 we show how utility functions can be created and
edited more easily by developers or expert-users. We simplify the creation process
in a way that even small mobile screens provide sufficient support.
Provide behaviour profiles. Multiple users with a single application or a single user with significant different habits might wish to individualise the applications' behaviour more precisely. In UC systems, developers can impossibly address all target users with a single adaptation profile, i.e. adaptive application behaviour. In Section 11.1.4 we introduce behaviour profiles based on a utility function policy with two distinct approaches: 1) multiple developer-defined static profiles and 2) multiple dynamic user-defined profiles. While the first approach provides developers with mechanisms to provide multiple utility functions during application development, the second approach is closely coupled with the modification of the reasoning policy. That means, when a user creates or alters a utility function, he can choose to save this new utility function as a behaviour profile. In both case, a user should easily switch between profiles at run-time.

Simpler adaptation reasoning specification. Reasoning policies like utility functions or goal policies are powerful, but are difficult to create and require internal knowledge on the application. Action-based policies are more straightforward, but struggle with large rule bases and inconsistencies in rules. By using fuzzy control we provide a hybrid solution between simple actions and complicated utility functions (cf. Section 11.2). A fuzzy knowledge base can be easily maintained by experienced users and its defuzzified result can feed into a utility function. Thereby, we retain the simplicity of rule-based knowledge bases and the expressiveness of utility functions while having at the same time the robustness of fuzzy logic and its ability to deal with uncertainty. Similar to our dynamic utility function solution, the user is able to define different fuzzy profiles to address multiple users or single users in multiple usage scenarios.

Learning of individual user behaviour. A self-adaptive application ideally embodies exactly the individual behaviour a particular user wants. Hence, we exploit a learning algorithm to map the user's individual requirements to the adaptation behaviour. In Section 11.3 we show how to learn application variants according to situations. Consequently, we abstract from context values and software components to allow for a user-oriented participation approach; both application variants and situations can be described in natural language terms. For the definition of situations, we employ the previously described fuzzy logic approach. Instead of reasoning over fuzzy metrics to be used in utility functions, we use fuzzy inference for defining situations based on specified context information. During a training phase, the user manually selects the application functionality he wants to use in a particular situation. There are no autonomous adaptations during this time. A learning engine samples the associated application variants according to the current set of situations. In operational mode, a k-nearest neighbour algorithm determines the best available application variant for the current situation according to the previously sampled training data.

7.3 Application Structure

The structural dimension addresses components, modules, and services of an application and the entities that provide them. Components or services can be either supplied by software developers or providers or they can be dynamically integrated from third-party
providers via standardised interfaces at run-time. In both cases the question is whether
the user or the system decides which of the components or services are actually used.
Moreover, the actual application structure and functionality could be changed by end-
users. The following mechanisms allow user participation in the structural dimension:

**Selection of functionality.** Software components modularise distinct functionality in
component-based software engineering. All components constitute the software
product, i.e. the application. Connections between these components determine the
software structure. Variability can be achieved by modification of the component
compositions (cf. Section 3.5). Developers define the available components and
compositions. Architectural constraints [78] may limit the number of application
variants at run-time, i.e. to avoid incompatible components. A user should be
able to select and deselect components and hence functionality for the adaptation
reasoning. Selection on the component-level is reserved for expert users. However,
appropriate abstraction mechanisms could shift the selection procedure to a higher
level of abstraction with simplifications for non-expert users.

**Selection of external services.** In the sense of service-oriented computing, different
service realisations and composition can create variability. Services can be
also realised by software components that are integrated into the application's
architectural model. Similar to the selection of native components, users should be
able to choose which services are integrated and whether the integration should be
done autonomously or with permission only. Services are more critical than native
components because they are mostly developed by third-party providers which
may harm the local software. The original application developers or the users can
hardly verify proper and safe execution of these services. Moreover, there may be
monetary costs involved with the integration of services.

**Modification of the architectural composition.** While the selection concepts work on
the developer-defined application structure and variability model, the next step
will be the modification of the structure and the variability model based on the
existing components. Expert users may re-arrange components, edit, remove, or
delete application variants. There is a challenge to provide this type of modification
to non-expert users because it requires expertise in software engineering and
application variability.

**End-user development.** End-user development of functionality has the strongest effect
in the structural dimension. Users can create their own components and component
compositions according to predefined rules and constraints. End-user development
requires comprehensive tools and workflow processes to allow the creation of
software artefacts without programming skills or advanced knowledge in software
engineering. Therefore, end-user development is typically used for simple tasks
that are easy to understand and have a high demand for individualisation by users.
8 Middleware-Based Self-Adaptation

Middleware-based self-adaptation is an external white-box approach to allow for architectural adaptation [184, 187–189, 113]. Our solution for user participation is embedded within the MUSIC approach [187]. In the following, we classify the MUSIC approach according to the taxonomy introduced in Chapter 3. A graphical representation of the taxonomy can be found in Figure C.1 in Appendix C. MUSIC employs application-layer adaptation for mobile computing applications in ubiquitous computing environments. Using component-based software design, it facilitates parameterisation and compositional adaptation techniques. Variability in parameterisation and component structures is modelled using a UML meta-model. The resulting variability model will be transferred into source code using model transformation in a MDD fashion and application functionality will be added subsequently. Planning, i.e. decision-making, is realised with a utility function policy. Although the decision-making is conceptually dynamic, i.e. open for run-time changes, there are no interfaces to change the specified utility function after deployment. The MUSIC approach does not consider uncertainty in any way nor does it provide interfaces for interacting with users or non-MUSIC systems. However, this applies to the majority of self-adaptation solutions.

![Figure 8.1: MUSIC system architecture.](image)

85
The MUSIC implementation (cf. Figure 8.1) builds on top of OSGi and declarative services [112]. Due to its mutual Java basis, it runs on various hardware platforms like Windows Mobile, ordinary notebook computers, or Android-based mobile devices. The middleware layer allows abstraction from the hardware, the operating system, and the adaptation process. Different developers can design applications and services as long as they comply with the modelling notation [155]. Each middleware instance can host adaptive applications and services. MUSIC services can be bound by other MUSIC middleware instances in a service-oriented fashion. Applications can integrate components from other applications within the same adaptation domain or components can be instantiated on different nodes. The context and adaptation middleware maps to a MAPE-K adaptation loop and represents the heart of MUSIC. It employs a plugin architecture to allow for adjustments in context sensing and adaptation reasoning.

Beyond the MUSIC adaptation middleware implementation, MUSIC further provides a systematic development methodology [155] to realise self-adaptive applications. A toolkit called MUSIC Studio [152] supports developers during the process. It includes tools for model transformation and validation in the MDD process as well as context simulation and visualisation. The development methodology and MUSIC studio play only circumstantial roles in our approach, instead we will focus on the actual adaptation approach.

In this chapter we will present the extended adaptation model that forms the basis for our participation concept. We will present details on the underlying conceptual meta-model, its terminology, and on the MUSIC adaptation approach. For a complete description of the MUSIC approach we refer to Khan [193] and the MUSIC documentation.

8.1 Variability

MUSIC combines a typing concept with component-based software engineering to create variability. The conceptual model depicted in Figure 8.2 illustrates plans as meta-concepts on the left side and concrete component implementations on the right side. We will now describe the terminology used for this variability concept.

**Component.** A component is an abstract description for encapsulated functionality in component-based software engineering [146]. Szyperski [145] defines a software component as “[…] a unit of composition with contractually specified interfaces and explicit dependencies where dependencies are specified by stating the required interfaces and the acceptable execution platform(s)”. A component can and typically will be connected with other components via provided and required interfaces. Further, a component can be structured [193], i.e. it may contain other components.

**Atomic Component.** An atomic component is a specialisation of a component that is not structured, i.e. it does not contain any other components.

**Composite Component.** A composite component is a specialisation of a component that has internal structures and is hence composed of other components.

---

1The MUSIC documentation: http://ist-music.berlios.de/site/documents.html (visited on 26/03/2014)
**Service Component.** A service component implements a component as a loosely-coupled service in a SOC fashion. Service components can be bound and used by other middleware instances. In combination with additional communication protocols (e.g., for service-level agreement), MUSIC implements a SOA. However, only MUSIC services can be shared between middleware instances.

**Component Type.** A component type defines an abstract set of functionality. Unlike a component, it does not provide any functionality by itself, but rather it requires a concrete implementation in form of a component. There can be more than one concrete realisation of a component. A component type further specifies a set of interfaces that are either required or provided.

**Application Type.** An application type is a specialisation of a component type. It is the most abstract component type as it defines functionality of an application. A single application type maps to a single application which can have different concrete realisations.

**Plan.** A plan is a self-contained middleware artefact that describes a component. It provides the required information to realise a particular component type. It comprises information on other included components, properties, and resource requirements. Both plan and component together establish the realisation of a component type. A plan is not a software variant. One plan can be used to describe components in many variants. Hence, a set of distinct plans describes a variant.

**Atomic Plan.** An atomic plan describes the realisation of an atomic component.

**Composite Plan.** A composite plan describes the realisation of a composite component and includes a description of the composed components.

**Service Plan.** A service plan describes the realisation of a service component. They include specific information required to integrate an external service, for example the host name, the port, or any parameters.
Application developers generate variability by modelling the application's structure as a UML diagram. The provided MUSIC UML profile (Appendix C) defines available concepts and dependencies among concepts. Compared to the conceptual meta-model depicted in Figure 8.2, the UML profile subsumes plans and components into so-called realisations. For the realisation of parameterisation, i.e. parametric adaptation, MUSIC introduces plan variants. Each plan has $1 \ldots n$ plan variants that describe different parameter settings within a component. During the model transformation into Java source code, plans and components are created separately from modelled realisations. The variability model does not include self-contained application variants, but instead variation points are inherited from the given typing concept. All possible application variants will be generated at run-time (cf. Section 3.5). In Part III of this work we will see examples of how variability is modelled.

Applications are installed as OSGi bundles into the middleware. These bundles contain the architectural description, the variability model, as well as the actual software components.

8.2 Self-Adaptation

Figure 8.3 depicts how MUSIC implements the MAPE-K adaptation loop. There are no distinct entities that match the MAPE-K concepts in a 1:1 scheme, instead the adaptation capabilities are distributed over the main entities Context Manager, Adaptation Controller, Plan Repository, and Configuration Controller. The different MAPE-K parts are depicted with circles at the MUSIC components in Figure 8.3. We will describe each of the MAPE-K components in the remainder of this section except the knowledge component. Knowledge is only implicitly available through the plan repository and the context manager. Plan structures provide knowledge on the application's variability and context plugins or reasoners may provide current and past context information, but there is no inference of new knowledge at run-time.

8.2.1 Monitoring and Analysis

The MUSIC middleware senses its environmental context via context plugins. A context plugin is the middleware equivalent to any type of information. This can be a hardware sensor like an accelerometer or a software sensor measuring message delay. Although technically similar, context reasoners derive high-level context information from other context plugins. For example, to provide accurate address information for a location, a context reasoner may use information provided by a GPS sensor context plugin and by a Wi-Fi localisation plugin.

Whenever an OSGi bundle is registered, containing plans and application types are added to the plan repository. Hence, the plan repository contains all available plans. Usually, there are not many changes in plans after they have been registered. Service plans for the integration of external MUSIC services are more dynamic. Services in ubiquitous computing environments may appear and disappear and hence the corresponding plans change accordingly.
The monitoring task is to observe the environment (context) and the performance of the adaptive application (self). Subsequently, the middleware analyses whether adaptation, i.e. re-configuration of the target application is required. Adaptation is triggered under the following circumstances:

- There is a change in the required context information. A context plugin or a reasoner decides if a change is significant and notifies the adaptation controller accordingly. Adaptation planning is only triggered if the change affects context information used by at least one of the running applications.

- There is a change in the plan repository. For example, when a MUSIC service has been discovered, all its provided plans were made available. Planning is only triggered if the changed plan is used by at least one of the running applications.

- An application has been removed or added.

Hence, the analysis task is carried out in two steps. First, by a context plugin or the plan repository that determine when to communicate the change and second, by the adaptation controller which determines which of the running applications require re-planning.

The monitored context information needs to be addressed and managed. Therefore, we use an ontology-based context model that provides the foundations for the context management. It defines the semantic concepts *Entity*, *Scope*, *Value Scope*, and *Representation*. An entity is a physical or logical entity of the world that is described by the context information, e.g. a smartphone or a person. The scope refers to the type of the provided information, e.g. the location. The scope can be stated more precisely with a value scope, e.g. the altitude value of a location. A representation describes how
the context information is internally structured and represented, for example Cartesian coordinates are used for the location. The ontology provides a common vocabulary to bridge semantic differences. Listing 8.1 presents an example how context information is addressed in MUSIC.

**Listing 8.1:** Example for addressing microphone context information with the semantic concepts *Entity, Scope, Value Scope, and Representation.*

<table>
<thead>
<tr>
<th>Entity</th>
<th>#Thing . Concept . Entity . Device</th>
<th><strong>this</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>#Thing . Concept . Scope . Microphone</td>
<td></td>
</tr>
<tr>
<td>Representation</td>
<td>#Thing . Concept . Representation . Microphone . Decibel</td>
<td></td>
</tr>
</tbody>
</table>

Within the context middleware, distinct context information is addressed as entity-scope-pairs. The representation is not used internally. For detailed information on the context middleware and the context model we refer to the work by Paspallis and Papadopoulos [201] and Wagner [154].

### 8.2.2 Planning

Adaptation planning in MUSIC is reactive to changes in the observed context. In contrast to goal-based planners, the MUSIC planner evaluates application configurations to the current context parameters in order to determine the best configuration for the current context parameter allocation. As this is not planning in the original sense, we will use the term *reasoning* from now on.

The reasoning works on so-called configuration templates. A configuration template is a middleware artefact created at run-time, representing one component realisation plan or a realisation variant for parametric adaptation. It also includes information on which MUSIC node the component will be instantiated. A configuration template hierarchy represents one application variant, i.e. each configuration template may contain other configuration templates according to the component composition scheme. The root template of each template hierarchy contains a plan describing the realisation of an application type. The configuration template structures are re-created every time the adaptation reasoning process is triggered.

The adaptation controller determines available reasoning plugins and selects one. The default reasoning plugin is a so-called *brute force reasoner*. The algorithm optimises all currently running applications. Therefore, it creates every possible combination of configuration templates from all applications. The evaluation of these templates requires more time and resource than some heuristic-based reasoner, but in return there is the guarantee that no application configuration is omitted. The adaptation middleware determines whenever reasoning is necessary, e.g. on context changes or newly discovered services in the environment. This can be any arbitrary moment despite if a user would agree or disagree on this particular moment. Due to its completeness and straightforward implementation, we will use the brute force reasoner as basis and as reference implementation for our user participation reasoning algorithms.

The brute force reasoning algorithm is listed in Algorithm 8.1. First of all, a template set is created for each of the running applications. A template set contains all possible application configurations for the particular application. For ease of understanding, the
Algorithm 8.1 The brute force adaptation reasoning algorithm.

```java
function REASON
    for all applications do
        create TemplateSet
    end for
    create TemplateSetEnumerator
    while TemplateSetEnumerator.hasNext do
        currentApplicationSet ← TemplateSetEnumerator.next
        CHECKRESOURCELIMITS(currentApplicationSet, availableResources)
        CHECKARCHITECTURALCONSTRAINTS(currentApplicationSet)
        weightedUtility ← 0.0
        for i ← 0, applications do
            template ← currentApplicationSet[i]
            rawUtility ← EVALUATE(template)
            weightedUtility ← weightedUtility + rawUtility * priority
        end for
        if weightedUtility ≤ bestUtility then
            bestUtility ← weightedUtility
            bestSet ← currentApplicationSet
        end if
    end while
    return bestSet
end function
```

To find the best global optimum, all template sets are combined in a combinatorial fashion. A template set enumerator iterates over these combinations sequentially, i.e. first, Template 1 from Application 1 and Template 1 from Application 2, then Template 1 from Application 2 and Template 2 from Application 2, and so on. Resource limits and architectural constraints are checked before each template is evaluated. The current application set is skipped if these checks fail. For the actual evaluation of a template, the utility function of each application is called. The utility function evaluates how good this configuration template hierarchy (i.e. this application variant) performs, given the current context parameters.

Listing 8.2: Utility function for the MUSIC pilot application Travel Assistant.

```java
double utility = uiQuality*0.4 +
    controllerQuality*0.2 +
    itineraryPlannerQuality*0.2 +
    outdoorMapProviderQuality*0.1 +
    metroMapProviderQuality*0.1;

//Higher utility with WiFi enabled indoor and WiFi disabled outdoor
if (locationType == LOCATION_OUTDOOR && wifiEnabled)
    utility = utility*0.75;
else if (!locationType == LOCATION_OUTDOOR && !wifiEnabled)
    utility = utility*0.5;

return utility;
```
The application developer has to specify the utility function which is a function of the current context and the plan properties determining the utility of an application variant. There is a single utility function for each application type. It is up to the developer how he implements the function, but it has to return a real value within the interval \([0, 1]\). Listing 8.2 provides an example of a utility function for the MUSIC pilot application *Travel Assistant* [42]. In the first part, the utility is calculated using a weighted sum of quality metrics. The quality metrics are calculated from plan properties and context values (not shown here). As there is only a single function per application and the application is a hierarchical composition of components, property values from other levels are aggregated by so-called property evaluators. In the second part, the utility is adjusted according to the current location of the user. This is a typical example of using rules (as IF-THEN-ELSE) to determine the utility of an application variant. The function will finally return a scalar value that assesses the particular application variant within the given context. This so-called *raw utility* is weighted according to the application’s priority \(p\) which is defined as:

\[
priority p_i \in \mathbb{R}; 0 \leq p_i \leq 1; \sum_{i=1}^{n} p_i = 1.
\]  

(8.1)

With the given priority, developers can privilege single applications over others during reasoning, i.e. component configurations used by multiple applications are prioritised. The weighted raw utilities of each application are added up to the weighted utility for the template set. The weighted utility for the template set is then compared to the best template set utility seen so far. The best application set is returned after all template sets have been evaluated.

Reasoning plugins like the brute force algorithm can work local or remote. Adaptation reasoning can be an expensive task regarding computation time and power consumption on mobile devices. Therefore, a MUSIC middleware instance can integrate a remote reasoning plugin provided by another middleware instance that is more powerful. However, for reasons of simplification we do not differentiate between local and remote reasoners for our user participation concept.

### 8.2.3 Execution

For re-configuration, the configuration controller makes use of computational reflection [93], i.e. components are instantiated and activated by their logical names instead of having a reference to a concrete class. The middleware cannot know about component classes created by application developers. However, it does know the component names described in the associated plans provided during installation of the OSGi application bundle.

The configuration of application variants (the best application set provided by the reasoning algorithm) involves several steps. The most important steps are:

*CreateInstances*. Creates instances of all components in the configuration template. In case a component is already instantiated, the existing instance will be used.
**RemoveInstance.** Removes all running instances of components that are not listed in the new configuration template.

**Connect.** Connects components within composite structures via ports according to the configuration template. Technically speaking, these components communicate via defined interfaces.

**Disconnect.** Disconnects all components of the previous application configuration. This step will always be executed even though the same components are used in the new configuration.

**Init.** Initialises components after instantiation and connection steps. Component developers can put their own initialisation procedure within the init method of a component. Only new components will run through the initialisation.

**Startup.** Starts the activity of a component. Developers should start the component activity within the startActivity method of the component. This step will always be executed after re-configuration.

**Finit.** The so-called *finit step* finishes a component before going over to remove the instance. At this point, a developer should make sure that the activity of the component is safely finished.

From a middleware perspective, components are loosely-coupled objects connected via ports. The configuration controller does not know if a component belongs to a particular application variant. Nor does it know which precise application uses this component or if it is re-used by other applications, too. This clean separation of concerns separates the adaptation capabilities from the application functionality. However, it makes user participation more difficult as users think on application-level and not on component-level.
9 Notification and Feedback

In Chapter 2 we elaborated the need for trust in adaptive applications. Trust is directly dependent on the dialogue principle *controllability*, i.e., users have more confidence in the application if they are able to manipulate the behaviour appropriately. In the previous chapter we introduced three different dimensions to manipulate a self-adaptive application. While the structural and behavioural dimensions require long-term modification which does not have to be done while a user focuses on his tasks, the temporal dimension addresses ad-hoc run-time modification while focusing on the current task. To modify the run-time behaviour, a user needs to know what is happening, i.e., when adaptations will occur or when his decision is required for a particular adaptation. A user needs to be informed and integrated within the adaptation loop. This information has a positive side-effect; it increases intelligibility and could also be used as explanations for why something is happening (cf. Section 2.5.3).

Adaptation middlewares like the MUSIC middleware do not provide any communication between an application and the middleware. Due to the de-coupled architecture and re-configuration scheme it is not necessary to have any communication. Although this is a sophisticated approach from a software engineering perspective, it prevents users from participating in the adaptation process and decreases intelligibility, controllability, and hence trust.

Subsequently we describe our notification and feedback patterns to realise bi-directional communication between the adaptation middleware and the user. *Notifications* are sent by the middleware. They contain information on past, present, or future adaptations. *Feedback* is given by the user in response to a notification or detached from notifications to influence the behaviour without prior impulse. First, Section 9.1 describes the different notification types in detail and then Section 9.2 introduces the middleware components to allow bi-directional communication between the user and the adaptation middleware.

### 9.1 Notifications

Notifications play a central role in our user participation concept as they deliver information effectively and permit users to control the application and the adaptation middleware while they are engaged in other tasks [95]. However, notifications require the user's attention, but attention is a limited resource in human-computer interaction [63]. Notifications interrupt the user from the current task and those interruptions can become disruptions and distractions from solving the tasks [9]. Although notifications cause interruption like autonomous adaptations do, they enforce trust and transparency when applied wisely and more importantly, they provide controllability.

According to McCrickard and Chewar [94], notifications can be designed regarding tradeoffs between the notification goals and attention cost of the notification. The
authors define three parameters: interruption, comprehension, and reaction. The parameters can be related to the design principles transparency and controllability, but allow more detailed design considerations.

![Diagram of notification types](image)

**Figure 9.1:** Schematic representation of three different notification types with feedback control. They differ in distraction, controllability, directness of control, and amount of displayable information [223].

Notification design depends very much on the type of computing system. We developed a notification design approach that is based on the notification pattern for smartphones from Hoober and Berkman [62] which has been modified regarding the parameters interruption, comprehension, and reaction. We distinct between three types of notifications [219, 223]: annunciator row, notification strip, and pop-up (cf. Figure 9.1). They differ in distraction, controllability, directness of control, and amount of displayable information. Each notification type contains the actual notification message and one or more elements for feedback control.

**Annunciator Row.** An annunciator row is also known as status bar in which the operating system places important system icons and status messages. The annunciator row needs two interaction steps for retrieving the notification message, but the actual notification is unobtrusive. The user can continue his work without being bothered by the notification. At the moment of notification only a short headline is visible. The message details will be displayed once the user drags or opens the notification. However, on most systems the length of the message details is still limited. It also depends on the operating system to what degree controllability can be achieved, but generally, no or only a single action can be taken from the annunciator row. For example, at the time we implemented the notification types, the Android API did not support buttons within the opened annunciator row. We designed a workaround that opens a new dialogue providing extended control options when the user clicks on the notification.

**Notification Strip.** A notification strip is a distinct area with a fixed location in the viewport that is dedicated to notifications. It is only visible when notifications

96 Notification and Feedback
are present, but they are still more distracting than a notification within the annunciator row. Parts of the current view are hidden beneath the notification strip. The notification strip should be placed at the bottom of the viewport to differentiate it from the status bar and the current view’s title. The notification message should fit into the notification strip, i.e. scrolling text must be avoided. The notification strip provides direct controllability. The information and the control elements are immediately visible. The amount of containing information is higher than with the annunciator row but less than with pop-ups. A notification strip may contain one or more control elements, but it has to deal with the limited space. There are no constraints given by the operating system, but it may burden developers with additional work because they have to develop the necessary concepts on their own.

**Pop-Up.** A pop-up is a new dialogue element on top of the current view. From the notification types it is the one with the highest distraction. Once the pop-up is shown, the user has to react on the notification message, because he cannot continue his current task without dismissing the pop-up. It should be either used if there is no good place for a notification strip, if the annunciator row of the operating system cannot be utilised properly, if the notification message is too long for the other two variants, or if we have to interrupt the user (should be used carefully). A pop-up provides the highest controllability as the user has to react on the notification. Moreover, we can place individual control options within the pop-up. Like the notification strip, the control is direct because the control elements are placed within the notification.

McFarlane [96] says that there is no best method for coordinating interruptions. The method is highly dependent on the current context of use, but the above propositions can be employed for immediate, negotiated, and scheduled interruption (cf. Section 2.5.2). In cooperation with feedback control elements users can also make changes to suggestions made in notifications. We refer to Kniewel et al. [222, 223] for more information on the design of notifications and design patterns for (self-)adaptive applications in particular.

### 9.2 Notification and Feedback Management

To achieve the necessary bi-directional communication between the user and the adaptation middleware, we introduce a notification manager and a feedback manager as new middleware components (cf. Figure 9.2). They follow the separation of concerns paradigm and fit as proxy components between applications and the middleware’s adaptation components. Both managers are implemented as OSGi services that connect via optional service interfaces to the other middleware components. Hence, the adaptation middleware is still fully functional without any of the new managers. Separating notification from feedback management allows more flexible configurations. We can image scenarios in which developers only require either notifications or feedback, or neither.

To allow users to be notified, the notification manager has to know where to send notifications. Therefore, applications require distinct components that work as notification clients. These can be any components and not necessarily user interface components. It would be rather wise to have one single notification receiver component
that delivers notifications to the currently available or appropriate user interface. A developer has to implement the `INotificationClient` interface or he can alternatively specify `receivesNotifications` for component realisations in the variability model of the application. Practically, any atomic or composite realisation can work as a notification client. The MDD process will then generate the necessary code fragments. During the `InitStep`, when configuring the application, the configuration manager registers all notification clients at the notification manager. This flexible approach makes notifications optional and allows downward compatibility to previous versions of the MUSIC middleware which do not support notifications. Moreover, changes at the application structure are not necessary. The notification management is only responsible for sending notification to notification clients. However, the application needs to know what kind of notification needs to be handled. It is then up to the developer what to do with the notification, especially what type of notification design should be used. Table 9.1 provides a list of notification reasons that are currently supported.

To enable the user to react on a notification or to modify the behaviour for long-term participation, a mechanism is required to influence the adaptation middleware. Thus, we introduce the feedback manager as a proxy component between application components

Figure 9.2: Integration of the notification and feedback managers as part of the new participation management within in the middleware.
and the adaptation middleware. Unlike the notification manager, which delegates notifications from the middleware to the user, the feedback manager takes feedback from the user to influence the decisions of the adaptation middleware.

A developer has to integrate access to the feedback manager by implementing the `IFeedbackComponent` interface. Alternatively, he can specify `sendsFeedback` in the atomic and composite component realisations of the variability model. However, the application should have at least one feedback component which should be a central component that is available in all application variants, too. Alternatively, if this cannot be achieved, multiple components can act as feedback components. Unlike the notification manager, the feedback manager must be accessible for the component to send the feedback to the middleware. To make the feedback manager accessible for an application component, we have to set a reference to the feedback manager in all feedback components during the InitStep of a component configuration (by the configuration manager).

<table>
<thead>
<tr>
<th>Notification Reason</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIGHER_UTILITY_PREFER</td>
<td>When the user did a manual adaptation by preferring a particular variant and there is another variant with a higher utility that the user should know about. For more information see Section 10.3 on manual adaptation.</td>
</tr>
<tr>
<td>HIGHER_UTILITY_FOCUS</td>
<td>When the current application variant is a user focus variant but another variant has a higher utility. For more details see Section 10.5 on user focus adaptation.</td>
</tr>
<tr>
<td>HIGHER_UTILITY_UNDO</td>
<td>When the user reverted an adaptation, a notification is sent if there is a variant with a higher utility available. For more information see Section 10.2 on post-adaptation control.</td>
</tr>
<tr>
<td>UTILITY_UNCLEAR</td>
<td>A notification can be sent if the current utility is unclear, i.e. not in favour of a particular variant.</td>
</tr>
<tr>
<td>UTILITY_DIFF_LOW</td>
<td>A notification can be sent if the current utility is only minimal lower or higher than the last utility which would not justify an adaptation (cf. Section 10.1 on pre-adaptation control).</td>
</tr>
<tr>
<td>RESUME_PAUSED</td>
<td>When the user paused the adaptation for a particular time, he is notified to resume the adaptation or if pausing should be extended. See Section 10.4 for more details on starting and stopping of adaptation.</td>
</tr>
</tbody>
</table>

Any middleware component can register itself via OSGi at the feedback manager as a so-called *feedback listener*. This means, whenever there is feedback from an application, the feedback manager will notify all its registered listeners and provides the actual feedback information. Then it is up to the notified listeners how to handle the specified feedback type. The currently supported feedback types are listed in Table 9.2. They are mostly used for temporal user participation (cf. Chapter 10) as this type of participation

9.2 Notification and Feedback Management 99
requires dynamic ad-hoc communication between the user and the adaptive system. For some special cases, we need communication between the notification manager and the feedback manager. First, when a user might want to disable notifications then the feedback manager has to tell the notification manager to turn them off and later back on. And second, when users select to pause adaptation for a particular time, the feedback manager creates a timer that will notify the user once it has expired.

9.3 Summary

User participation for self-adaptive software requires bi-directional communication between the user of an adaptive application and the adaptation middleware. In this chapter we presented two essentials concepts to achieve this type of communication. First, we adapted different types of notifications to be used in combination with adaptive software. Notifications inform users on ongoing adaptation actions and allow them to react accordingly. Depending on the notification type, the amount of information, the degree of controllability, and the obtrusiveness varies. And second, we introduced the notification and feedback managers that extend the adaptation middleware with capabilities for notification and feedback management. The notification and feedback management is independent of the number of users and can handle multiple running applications at the same time. The subsequently described temporal and behavioural participation dimensions require a working notification and feedback management.
Table 9.2: Types of feedback.

<table>
<thead>
<tr>
<th>Feedback Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP_ADAPTATION</td>
<td>Stops adaptation permanently for the application that initiated the feedback.</td>
</tr>
<tr>
<td></td>
<td>See Section 10.4 for more details on starting and stopping of adaptation.</td>
</tr>
<tr>
<td>START_ADAPTATION</td>
<td>Starts adaptation permanently for the particular application after it has been stopped before.</td>
</tr>
<tr>
<td>TOGGLE_ADAPTATION</td>
<td>Activates or deactivates adaptation globally depending on the current state. This applies for all running applications.</td>
</tr>
<tr>
<td>PAUSE_ADAPTATION</td>
<td>Pauses adaptation for a specified interval of time for the application that initiated the feedback. The middleware automatically resumes adaptation after the time expires.</td>
</tr>
<tr>
<td>PAUSE_ADAPTATION_ALL</td>
<td>Pauses adaptation for all running applications for a specified interval of time.</td>
</tr>
<tr>
<td>PREFER_PLAN</td>
<td>Prefers a particular application plan. For more information see Section 10.3 on manual adaptation.</td>
</tr>
<tr>
<td>UNPREFER_PLAN</td>
<td>Removes the preferable setting of a plan in case it has been set before.</td>
</tr>
<tr>
<td>UNDO_ADAPTATION</td>
<td>Reverts the most recent adaptation. For more details see Section 10.2 on post-adaptation control.</td>
</tr>
<tr>
<td>IGNORE_UNDO</td>
<td>If an adaptation has been undone before, this type of feedback will return to normal operation mode (cf. Section 10.1 on pre-adaptation control).</td>
</tr>
<tr>
<td>IGNORE_USER FOCUS</td>
<td>Ignores a predefined user focus adaptation setting if not applicable for the current situation. For more details see Section 10.5 on user focus adaptation.</td>
</tr>
<tr>
<td>TOGGLE_USER_FOCUS</td>
<td>Switches between normal brute force reasoning and user focus reasoning.</td>
</tr>
<tr>
<td>TOGGLE_PRE_ADAPTATION</td>
<td>Toggles standard pre-adaptation control, e.g. notification of users on unclear utilities.</td>
</tr>
<tr>
<td>TOGGLE_TRAINING</td>
<td>Switches between training and operational mode of the learning-based adaptation reasoner (cf. Section 11.3 on learning-based adaptation reasoning).</td>
</tr>
<tr>
<td>RESET_KNOWLEDGEBASE</td>
<td>Resets the knowledge base of the learning-based adaptation reasoner in case the learned behaviour is not appropriate anymore.</td>
</tr>
<tr>
<td>TOGGLE_NOTIFICATIONS</td>
<td>Depending on the current state, notifications are switched on or off.</td>
</tr>
</tbody>
</table>
10 Temporal User Participation

This chapter covers the mechanisms from the temporal participation dimension. We start with pre-adaptation and post-adaptation controllability followed by manual adaptation and disabling adaptation as straightforward concepts for user participation. This chapter concludes with the more advanced concepts for user focus reasoning and user interaction activity detection. All mechanisms address the short-term ad-hoc interaction of users with self-adaptive applications. Although each of the mechanisms is described in an isolated manner, they all can be combined to fulfil individual user requirements.

10.1 Pre-Adaptation Controllability

Pre-adaptation controllability provides the user with control before an adaptation is executed, i.e. the user can decide if the system should adapt beforehand. The system decides how to adapt, but it does not perform the adaptation autonomously. The reasoning algorithm determines the best set of configuration templates and compares the current application configuration with the new configuration. In case of re-configuration, the adaptation reasoner will not configure components in the execution phase, instead it first checks if the user has to be notified and the configuration phase is blocked, otherwise the adaptation is executed autonomously. If the user had to be notified, the decision is sent to the adaptation reasoner via the feedback manager. In case the adaptation is accepted, the adaptation reasoner proceeds as it would be without any controllability. When he declines the adaptation, the adaptation reasoner withdraws the new configuration and marks the application as not changed. We provide two configurations for pre-adaptation controllability when the user has been notified:

1. No autonomous adaptation after timeout: If the user does not make any decision within a specified period of time, the adaptation reasoner assumes a declined adaptation and keeps the current configuration. The user must then adapt manually (cf. Section 10.3).

2. Autonomous adaptation after timeout: If the user does not make any decision within a specified period of time, the adaptation reasoner proceeds with the configuration for the new configuration template autonomously.

Notifications are sent depending on the configuration of the adaptation reasoner: either it is configured to send notifications with every adaptation or when the difference in utility between the new and old configuration template is below a specified threshold, i.e. when

\[ \Delta_{\text{notify}} \leq |\text{new}_\text{utility} - \text{old}_\text{utility}|. \]  

(10.1)
The parameter $\Delta_{\text{notify}}$ is specified by the developer or maintainer. This approach is based on the assumption that small differences between utility values express only little improvement which does not justify the consequences of an autonomous and unannounced adaptation. A third option for pre-adaptation control can be used when the utility is unclear and none of the evaluated application variants is significantly better than any other, i.e. if the set of all application variants $\{V_1, V_2, \ldots, V_n\}$ is denoted as $\mathcal{V}$ and the following holds:

$$\forall V_i, V_j : |\text{utility}_i - \text{utility}_j| \leq \Delta_{\text{small}}; \ i \neq j; V_i, V_j \in \mathcal{V}. \quad (10.2)$$

Again, the parameter $\Delta_{\text{small}}$ is specified by the developer or maintainer.

Re-adaptation can be triggered with every change in the context, i.e. adaptations may occur with a high rate. Many repeating notifications do not increase user acceptance, but rather they can be as annoying as sudden adaptations are. Therefore, we use notification rate limitation for each application, based on a sliding window: if the middleware has sent a number of $n$ notifications to the user within a time period $t$, it will not send the notification. In this case, all adaptation actions are dismissed.

### 10.2 Post-Adaptation Controllability

Post-adaptation controllability provides the user with control after an adaptation has been executed, e.g. accepting or declining the adaptation in the simplest case. If adaptations occur which the user did not want in the current situation, he should be able to undo such adaptations. Adaptation middlewares like the MUSIC middleware are stateless and do not keep any history of adaptations executed before. During each reasoning process, the available configuration templates are evaluated against the current context only. With every evaluation the current situation is considered as independent of any evaluations and adaptations happened before. Only the current and the next configuration templates are known. Therefore, if the user decides to undo an adaptation it will contradict the decision of the adaptation middleware; its last decision to be precisely. Reverting adaptations requires restoring the previous state of an application. In our solution we provide the component structures and leave the actual state transfer of their functionality to the developer. For example, he must store the current configuration of a user interface within the $\text{finit}$ configuration step and restore the configuration within in the $\text{init}$ step.

The adaptation reasoner has to know that reverting the adaptation is a modification made by the user and it should not be overwritten during the next reasoning cycle. One possible solution is using the same technique as with manual adaptation, i.e. to prefer a plan that the middleware will consider during future reasoning (cf. Section 10.3). The preferable plan would be one of the last used plans, but we do not know which plan within a configuration template to take for the preference. Of course, we could prefer all plans in this template using conjunctive plan preferences, but this would bring large reasoning overhead and it could interfere with previously preferred plans. Hence, we implemented an approach in which the next configuration template is overwritten by the currently instantiated configuration template. By simply setting the next template this way there is no need to reason again. Furthermore, we have to make sure that
this template is not overwritten during a new evaluation cycle. Therefore, we mark the application as _undone_ which will prevent the adaptation reasoner from setting any new template.

Once an adaptation has been reverted, no further autonomous adaptations are made by the adaptation controller. However, we do not disable reasoning to allow the user to return to autonomous behaviour. If such a template evaluation results in a higher utility than the currently active template (which is very likely as the user has reverted the best template before), a notification can be sent to the user stating that there is a better configuration template available. As with pre-adaptation controllability, the developer has to specify how much higher a utility should be for notifications, i.e. when

\[
\Delta_{\text{notify}} \leq \text{new}_\text{utility} - \text{old}_\text{utility}. \tag{10.3}
\]

A user may decide to ignore the previously undone operation by accepting a notification and sending feedback. In this case the _undone_ flag is removed from the application and reasoning is triggered again to select the currently best template.

### 10.3 Manual Adaptation

A user should be able to change application variants by simply interacting with the application, i.e. by pressing a button, selecting a menu entry, or performing other actions. We call this _manual adaptation_ as it is not done by the adaptation middleware autonomously. Instead, the user decides what will happen next, i.e. he overrides the system's default behaviour for this particular moment.

We assume that the user selects a particular function or a configuration which is realised in its own component described by a plan (either composite or atomic). We cannot simply instantiate a configuration template containing this plan because the autonomous adaptation reasoning process will re-instantiate the previous template in the next adaptation cycle as we have not changed the actual decision base.

We extend the plan structure by a _preferable_ attribute indicating that this particular plan is currently preferred by the user. A plan that is preferred by the user has to be included in the configuration template to be instantiated. A configuration template which includes at least on preferred plan is called _preferred template_. However, as we do not want to lose autonomous behaviour completely, the adaptation middleware has to make sure that the user is not tied to preferred templates only. For example, we can notify the user in case there is a non-preferred configuration template which has a much higher utility than the currently preferred template.

Manual adaptation requires feedback from the application to the middleware. To make use of plan preferences, a developer has to integrate access to the feedback manager by implementing the _IFeedbackComponent_ interface in his component. Alternatively, he can specify _isFeedback_ in the realisations of the application's variability model (cf. UML modelling profile in Appendix C). The MDD process will then generate the necessary code fragments. The feedback manager takes the component type and plan name to find the corresponding plans in the local plan repository and marks them as _preferable_. Then,
it updates the plan in the local plan repository. This plan change triggers adaptation in
the middleware and the reasoning algorithm will consider the preferable attribute.

A developer can choose between different types of preference: conjunctive multiple
plans, disjunctive multiple plans, or a single exclusive plan. A conjunctive combination
makes sure that only templates can be selected which include all chosen preferred plans.
With a disjunctive combination of plans, the resulting best configuration template does
not necessarily need to include all of the preferred plans. When using exclusive plan
preferences, all previous preferences are removed before setting the new preference.

Algorithm 10.1 Brute force reasoning with user-preferred plans (manual adaptation).

```
function REASON
    for all applications do
        create TemplateSet
    end for
    create TemplateSetEnumerator
    while TemplateSetEnumerator.hasNext do
        currentApplicationSet ← TemplateSetEnumerator.next
        CHECKRESOURCELIMITS(currentApplicationSet, availableResources)
        CHECKARCHITECTURALCONSTRAINTS(currentApplicationSet)
        weightedUtility ← 0.0
        for i ← 0, applications do
            template ← currentApplicationSet[i]
            rawUtilities[i] ← EVALUATE(template)
            weightedUtility ← weightedUtility + rawUtilities[i] * priorities[i]
        end for
        if weightedUtility ≥ bestWeightedUtility AND preferable then
            bestWeightedUtilities ← rawUtilities
            bestSet ← currentApplicationSet
        end if
        if weightedUtility ≥ maxWeightedUtility then
            maxWeightedUtilities ← rawUtilities
        end if
    end while
    return bestSet, bestUtilities, maxUtilities, priorities
end function
```

The reasoning algorithm for manual adaptation is based on Algorithm 8.1 in Section 8.2.2.
However, during the reasoning process only templates will be selected that contain at
least one plan with the preferable attribute set. The extended algorithm is shown in
Algorithm 10.1. First, the algorithm checks if any of the configuration templates contains
a preferable plan. Then, the highest value of those templates is stored as best_utility. To
be able to return to the autonomous behaviour, all other available templates are also
evaluated. The highest utility of all templates is stored in max_utility. Hence, we could
notify the user if there is a much better plan than the currently preferred plan. The
developer has to specify how much higher a utility should be for being notified, i.e. when

\[ \Delta_{\text{notify}} \leq \text{max\_utility} - \text{best\_utility}. \]  

(10.4)
As the reasoning algorithm works on a configuration template basis, it needs to know when to consider preferable plans. Hence, it is first checked whether a configuration template hierarchy contains a preferable plan. There is a chance that the first template hierarchy does not contain such plan but the second or third does. Thus, the algorithm must determine if any of the configuration templates to be evaluated contains a preferable plan. One solution would be to evaluate all templates twice. First, it is checked if any of the templates contain a preferable plan and second, for the actual evaluation. As this approach is not very efficient we decided to integrate the information whether a template hierarchy contains a preferable plan or not in the template set iterator. The template set iterator is created by the template builder prior reasoning and combines all possible application variants from multiple applications (cf. Section 8.2.2). It recursively checks if any of the contained plans has the `preferable` attribute set.

The adaptation reasoning returns to normal mode when no plan is preferred anymore. The `preferable` attribute is removed using a feedback manager API call (un-prefer), e.g. when the user accepts a message from the notification manager regarding a better suitable application variant for the current situation. The feedback manager notifies the plan repository due to plan changes, which again causes a re-adaptation.

Algorithm 10.1 supports disjunctive multiple plans and single exclusive plan preference. We will use conjunctive plans in the learning-based reasoning algorithm where we need plan preferences during training (cf. Section 11.3). It is also notable that preferring a plan affects all applications using this plan as plans are not specific to applications.

### 10.4 Starting and Stopping Adaptation

Users should have the ability to disable the adaptive behaviour and enable it later at any moment. This is what we call *toggle adaptation*. It increases controllability and trust into adaptive applications as users are in charge of a central component. Adaptation frameworks or middlewares usually do not foresee a non-adaptation mode as their intention is to provide adaptation. Even with disabled adaptation capabilities, the application has to remain usable and should provide some kind of functionality for the user. In the worst case (from the adaptation perspective) a user disables adaptive behaviour completely and makes use of manual adaptation (Section 10.3) solely. This implies two assumptions: first, there is an application configuration available also when adaptation is turned off and second, the user can switch configurations when adaptation is turned off.

We provide two solutions for disabling self-adaptive operations by users:

1. Use the last configured application variant.

2. Configure a default application variant that includes basic application functionality.

It depends on the system maintainer which solution will be used. In both cases, the user can control the activation and deactivation from the application’s user interface for only this application or for all applications running on the same middleware instance. He can either stop self-adaptive operation permanently or pausing it for a given period of time. When selecting to pause adaptation, the user will be asked at the end of the period if he wants to resume self-adaptive operation. He will receive the notification on the same
application from which he initiated the pause. Pausing adaptation can also be used by software routines to prevent running software transactions (e.g. database operations) from being interrupted by adaptation. Although the MUSIC component model allows suspension of components, which can be used to wait for running transactions, the suspension process is supposed to be quick. Hence, longer transactions must be interrupted by the middleware or the adaptation will be executed much later when it is likely to be not optimal anymore.

When using the last configured application variant, stopping or pausing self-adaptive operation stops the adaptation controller from adapting an application. The currently instantiated application configuration is not altered. Resuming self-adaptive operation starts adaptation handling in the adaptation controller again.

Simply keeping the current configuration is the easiest way to provide a configuration of the application when adaptation is turned off. But, the current configuration might be very specific for the current situation and the user might feel “stuck” in such a configuration. Further, it does not solve the problem when adaptation is disabled during the start of an application. Which configuration should be taken then? Hence, we propose the concept of a default configuration. A default configuration is a developer-defined application variant that is used when adaptation is disabled. We recommend including the basic application functionality in this application variant. But, a developer does not define application variants, rather he defines component types and realisations with implicit variation points (cf. Section 8.1). Our solution is a mechanism to define default behaviour over component boundaries.

A developer can mark component plans that belong to the default variant directly in the variability model. Component realisations can optionally have an attribute isDefault with \( \text{isDefault} = \{0, 1\} \). Components with this attribute set belong to the default variant. However, we have to rely on the developer’s skills that he only marks components that clearly constitute a single variant. If he forgets to mark a component, no default variant exists. Therefore, we extended this scheme by calculating a default degree for all configuration templates.

The default degree determines for each configuration template hierarchy the ratio of realisations marked as default compared to the total number of realisations. We further consider the default realisation’s level in the template hierarchy. The closer a default realisation is located to the root of the hierarchy, the more important it is for the calculation of the default degree. It is assumed that realisation plans at lower levels of the hierarchy determine only peripheral changes in the application functionality whereas realisation plans on the upper level define main application modes that should not be outweighed by low-level realisations. Like utility values, the default degree is normalised to the interval \([0, 1]\). Algorithm 10.2 shows the recursive calculation of the default degree.

The default template reasoning algorithm is listed in Algorithm 10.3. Again, it is based on the brute force reasoner that evaluates all available application configurations. However, when in default reasoning mode this algorithm does not evaluate a configuration template’s utility function, instead the default degree of each template hierarchy is determined. The default degree relates proportional to utility values, i.e. the higher the default degree, the better the application set. The algorithm will return the best application set regarding its default degree. Like the original brute force algorithm it will return the first best application set in case multiple application sets have the same default
degree. For example, default reasoning mode can be enabled when the user stops the self-adaptive behaviour of an application. Then the default template reasoning algorithm will determine the default template. Once the default template has been determined, no further reasoning actions will be done until the user resumes to self-adaptive operation.

**Algorithm 10.2** Calculation of the default degree of a configuration template hierarchy.

```plaintext
function CALCULATEDEFAULTDEGREE(Template)
    defaultDegree ← isDefault
    if childTemplates = empty then
        return defaultDegree
    end if
    for all childTemplates do
        defaultDegree ← defaultDegree + calculateDefaultDegree(child)
    end for
    return defaultDegree / (childTemplates.size + 1)
end function
```

Figure 10.1 illustrates the calculation of the default degree with a simple example based on the MUSIC modelling notation. This example consists of three application variants i.e. configuration template hierarchies. Atomic realisations 2 and 3 have been marked as isDefault, as well as composite realisation 1. Application variant 2 consists only of component type realisations (plans) that are marked as default and hence comes up with a default degree of 1.0. The full application model for this example is available in Section D.2 of the appendices.

![Figure 10.1: Three application variants and their default degree.](image)

The default application variant may also be instantiated if no other application variant could be found, for instance when resource constraints prevent the selection. Therefore, any reasoning algorithm can be extended by a second default reasoning phase which is triggered when no application variant was found. Before, the adaptation reasoner in MUSIC aborted application configuration when no valid configuration could be found. A developer must consider this case explicitly in the utility function.
Algorithm 10.3 Brute force reasoning in combination with default template reasoning.

function REASON(DEFAULT_MODE)
    for all applications do
        create TemplateSet
    end for
    create TemplateSetEnumerator
    while TemplateSetEnumerator.hasNext do
        currentApplicationSet ← TemplateSetEnumerator.next
        CHECKRESSOURCESLIMITS(currentApplicationSet, availableResources)
        CHECKARCHITECTURALCONSTRAINTS(currentApplicationSet)
        weightedUtility ← 0.0
        for i ← 0, applications do
            template ← currentApplicationSet[i]
            if defaultMode then
                rawUtility ← GETDEFAULTDEGREE(template)
            else
                rawUtility ← EVALUATE(template)
            end if
            weightedUtility ← weightedUtility + rawUtility * priority
        end for
        if weightedUtility ≤ bestUtility then
            bestUtility ← weightedUtility
            bestSet ← currentApplicationSet
        end if
    end while
    return bestSet
end function

10.5 User Focus Adaptation

In Section 3.6 we defined weak and strong adaptations. Weak adaptations have only little impact on the system and are less complex, for example changing run-time parameters. In contrast, strong adaptations involve major system changes like replacing software components. Especially strong adaptations likely have a negative impact on the user as they may be distracting and disturbing from the user’s current task. Pre-adaptation and post-adaptation control help to lower the negative impact but sending notifications with every single adaptation may also distract the user from its current task. Therefore, we extend the concept of weak and strong adaptations in regard to user interfaces changes and user distraction that include both compositional and parameter adaptation [218]:

1. High user-related adaptations: components directly related to the user interface are modified. Consequently, the functionality and the interaction flow are very likely to be changed.

2. Low user-related adaptations: components are modified according to context changes. The underlying functionality may change but it is not as likely as with strong adaptations.

110 Temporal User Participation
3. **No user-related adaptations**: components are replaced to achieve a better QoS. The actual functionality of components does not change and the user will not notice any change in the available functionality, but the performance of the currently available functionality may improve.

Based on this classification we can provide a more fine-grained adaptation concept in respect to user-related adaptations. We introduce the concept of a *user focus* to avoid user distraction and loss of control by self-adaptive middleware operations [219]. A user focus addresses **high user-related adaptations**. Hence, we define a user focus as a concept representing the current intention of use and comprising a collection of associated functions and software components that are in the user’s perceptual focus while following his current task. It is crucial to understand that the user’s intention of use directly relates to the adaptive behaviour of the application (cf. Figure 10.2).

![Figure 10.2: Interplay between the user and the adaptive system.](image)

While the user is following his current task with a particular intention, a self-adaptive middleware captures the context through sensors and decides which application realisation will be the best for this particular situation. A wrong decision of the middleware may yield in increased user distraction and anxiety (e.g. when performing high user-related adaptations) as the new application realisation does not necessarily match the user’s intention of use. For the adaptation middleware it is hard to detect when it is a good moment for adaptation, i.e. to avoid user distraction and loss of control.

User focuses are disjunctive; a user can never be in two focuses at the same time as he cannot have multiple intentions of use in a single moment\(^1\). A user focus does not describe an interaction with the application; rather it can be seen as a logical part of the application that is in the perceptual focus of the user in a particular moment in a particular context of use. We define user focuses on realisation plan level, i.e. any plan and any combination of plans can possibly represent the user’s perceptual focus. Any plan that is not a user focus plan is called background plan.

---

\(^1\) Naturally, the statement that a user can never be in two focuses at the same time holds only for devices with a single interaction interface. With multi-screen or even multi-modal interaction, the concept of user focuses needs to be extended.
Both user focus and background plans directly map to the variability scheme provided by MUSIC (cf. Figure 8.2), i.e. they can be either composite or atomic plans. Hence, user focuses of composite plans may have further nested user focus plans according to the typing scheme. To avoid sudden user distraction, an adaptation middleware should not be allowed to autonomously adapt any components described by a user focus plan; rather it should notify the user in case there is a better variant available and let the user decide what to do next. In contrast, components described by background plans can be reconfigured by the middleware without further request. An application developer has to identify all direct and (if applicable) nested user focuses during requirements elicitation phase. For each user focus realisations the attribute isFocus with isFocus = \{0, 1\} needs to be set in the variability model. For example, in Figure 10.3 the Atomic Realisation 2 has become a user focus plan. Once identified and marked in the variability model, the adaptation middleware can consider user focus plans at run-time.

The benefit of the user focus concept comes during execution, when components grouped into a user focus will not be adapted automatically by the middleware. The user stays in the current focus and would be notified if there were any better realisation plan available that is outside the current user focus. Then it is up to the user whether the middleware should adapt or not (e.g. using pre-adaptation controllability).

To leave a user focus configuration, the user has three alternatives depending on the configuration of the adaptation controller:

1. **Pre-adaptation controllability** (cf. Section 10.1): choose \(\Delta_{notify} \leq max\_utility - best\_utility\) with a default value of \(\Delta_{notify} = 0.5\). That means, the user is notified of a better application in spite of the current focus variant.

2. **Post-adaptation controllability** (cf. Section 10.2): choose \(\Delta_{notify} \leq max\_utility - best\_utility\) with a default value of \(\Delta_{notify} = 0.9\). That means, despite a set user focus, the adaptation controller will adapt the application to a non-focus/different focus variant, but providing the undo option.

3. **Manual adaptation** (cf. Section 10.3): the user must switch manually to a different application variant by selecting the respective function.

To realise user focus adaptation, a simple approach is to check if the new application set (determined by the reasoning algorithm) contains the same user focus plans as the currently instantiated application set. If yes, the adaptation reasoner should proceed and adapt accordingly. If not, it should keep the current application set. This check is fairly simple and effective, but it has a negative side-effect which comes due to the nature of brute force reasoning in MUSIC: every available configuration template is evaluated and has a utility according to the application-specific utility function. If we prevent the adaptation engine from adapting, all other templates are still evaluated. Any other template \(t_x\) (without the particular focus plan) than the currently active template has the highest utility. This is not a problem as long as there is no template \(t_y\) including the focus plan, but having a lower utility than the variant without the focus plan. In both cases, an adaptation reasoning algorithm like the brute force reasoner determines the non-focus variant as the best variant and will stay with the non-optimal current focus variant. This problem is illustrated in Figure 10.3. Variant 1 with the highest utility of 0.8 is a non-focus template hierarchy whereas Variant 2 and 3 contain a user focus plan. Variant 2 is the currently active variant. However, Variant 3 is better rated for the
current situation than Variant 2 and should be instantiated. But, the brute force reasoner will always determine Variant 1 as the best variant and as this one does not contain the particular user focus plan, the currently active variant will be kept. Again, the full application model for this simple example is available in Section D.2 of the appendices.

We developed a reasoning algorithm that is able to consider a user focus during the adaptation decision similar to the plan preference algorithm with manual adaptation (cf. Algorithm 10.4). Based on the brute force reasoning algorithm, it first checks if the currently active configuration template hierarchy contains a user focus plan. If yes, the reasoning algorithm will determine a new application set that also contains the user focus plan. Unlike the previous presented reasoning algorithms, the user focus algorithm needs knowledge on the currently instantiated configuration templates. Like with plan preferences in manual adaptation, we have to evaluate application sets containing the particular user focus and all other application sets simultaneously. We store the best_utility for the configuration template hierarchies containing this particular user focus plan and also the max_utility over all other hierarchies. The application set with the highest best_utility is chosen for user focus adaptation.

When combining user focus reasoning with one of the previously presented mechanisms, we have to consider some special cases. When using manual adaptation in combination with user focus reasoning, the preferred plans from manual adaptation are prioritised over the user focus settings. This is particularly important when users want to leave a user focus configuration. When a user decides to remove the preferable setting of a plan (un-prefer), unexpected behaviour may occur if the previously preferred plan is a user focus plan. Then, no adaptation would happen even if this were expected by the user (he accepted the notification before and decided to un-prefer a plan). As this behaviour is probably not desired, the adaptation controller is instructed to ignore the user focus settings for this one reasoning cycle. The problem with the specific MUSIC implementation is the loosely attached user focus reasoner that simply iterates over the configuration templates and knows only required and available resources but nothing

**Figure 10.3:** Non-optimal decision with the standard brute force reasoner in combination with user focus plans. Plans without the isFocus attribute are called background plans and can be included in autonomous adaptations without further limitations.
Algorithm 10.4 The user focus adaptation reasoning algorithm.

function REASON
    for all applications do
        create TemplateSet
    end for
    create TemplateSetEnumerator
    while TemplateSetEnumerator.hasNext do
        currentApplicationSet ← TemplateSetEnumerator.next
        CHECKRESOURCIELIMITS(currentApplicationSet, availableResources)
        CHECKARCHITECTURALCONSTRAINTS(currentApplicationSet)
        weightedUtility ← 0.0
        userFocus ← hasUserFocus(runningApplicationSet)
        for i ← 0, applications do
            template ← currentApplicationSet[i]
            rawUtilities[i] ← EVALUATE(template)
            weightedUtility ← weightedUtility + rawUtilities[i] * priorities[i]
        end for
        if weightedUtility ≥ bestWeightedUtility AND userFocus then
            bestWeightedUtilities ← rawUtilities
            bestSet ← currentApplicationSet
        end if
        if weightedUtility ≥ maxWeightedUtility then
            maxWeightedUtilities ← rawUtilities
        end if
    end while
    return bestSet, bestUtilities, maxUtilities, priorities
end function

on why adaptation happened. To adjust the reasoning process depending on different adaptation options (e.g., to ignore the user focus), we make use of the adaptation reason information provided for every application. The reason is set by the adaptation controller after receiving feedback from the feedback manager. In case the template iterator does not contain a preferable plan and the adaptation reason is to ignore the user focus, all templates are evaluated equally. This approach allows the reasoner to select the best available configuration template as this was the reason for the original notification which the user accepted.

Reverting adaptations due to post-controllability affects user focus reasoning, too. Once an adaptation has been reverted, the adaptation middleware will not perform adaptations autonomously neither for templates containing user focus or background plans.

10.6 Detecting the User’s Activity

The user focus concept presented in the previous section is rather static in regard to the actual run-time intentions and tasks of a user. It assumes that the developer-defined user focus plans match to the situation during application execution. What if we could
determine if the user would be currently engaged with the mobile device and the task and adjust the adaptive behaviour according to the degree of interaction activity?

<table>
<thead>
<tr>
<th>Class</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>no activity (n)</td>
<td>display off</td>
</tr>
<tr>
<td>undetermined activity (u)</td>
<td>display on</td>
</tr>
<tr>
<td></td>
<td>starting/stopping applications</td>
</tr>
<tr>
<td></td>
<td>switching applications views</td>
</tr>
<tr>
<td></td>
<td>low touch screen activity</td>
</tr>
<tr>
<td>high activity (h)</td>
<td>key lock off</td>
</tr>
<tr>
<td></td>
<td>active telephone call</td>
</tr>
<tr>
<td></td>
<td>medium to high touch screen activity</td>
</tr>
</tbody>
</table>

Our assumption is that we can achieve less obtrusive and distractive adaptation behaviour by respecting the user's current interaction activity. We define interaction activity as actions performed by a user with the device to achieve his current tasks. Such actions can be explicit user actions like clicking, touching, scrolling, reading, entering text, phoning, or pressing a button. These actions can be also implicit statuses determined by the device. For example, having the device in the pocket or having the device locked. Our goal is not to detect what type of activity the user is currently pursuing, e.g. walking, running, talking, standing, etc. We are interested in the interaction activity, i.e. if the user is currently interacting with the device and to what degree.

On the one side, the degree of interaction activity can be used to determine whether an adaptation will likely disturb the user or not. And on the other side we can use information on interaction activity to decide what type of notification we have to send to the user. For example, when a user is currently talking to someone on the phone, we should not adapt silently, rather we should notify, either with a sound, a notification message, or both – depending on the priority of the adaptation. Therefore, we define the following three classes of interaction activity:

1. **No activity (n)**: No activity describes no interaction with the device at all, e.g. when the display is turned off and the user is not making a call.

2. **Undetermined activity (u)**: The undetermined activity class describes low device activity which cannot be certainly correlated to actual user activity, e.g. changes in running applications.

3. **High activity (h)**: With high activity we know for sure that the user is currently interacting with the device, e.g. due to high touch screen activity.

Although continuous activity values could be derived from the sensors, these discrete classes are sufficient to adjust adaptations based on interaction activity. Table 10.1 lists the assignment of actions (determined by sensors) to their respective activity classes. From these assignments of actions to classes we use a rule base that determines the degree of activity according to sensor inputs. At first, we used the highest class if multiple

10.6 Detecting the User's Activity  115
rules apply to the current situation. For example, when there is low touch screen activity and the key lock is off, the resulting degree of activity is *high*.

![Diagram](image)

(a) The user starts an application.

![Diagram](image)

(b) The user is reading information provided by the application.

![Diagram](image)

(c) The user is entering text.

![Diagram](image)

(d) The user has sent the text, waits for the application's result and verifies it.

**Figure 10.4:** Determining a user's interaction activity in a four-step process: 1) two sensors determine their activity class, 2) the maximum sensor value is used in the sliding window, 3) the majority value from the sliding window is determined, and 4) the maximum of sensor values and the sliding window determines the final activity class.
However, due to the highly dynamic input sensors, the determined activity class changed quickly. To get a stabilised activity classification output, we use a four-step approach. In the first step, user activity is directly determined for each sensor according to the underlying rule base. Next, a sliding window mechanism determines the activity class for a given (time) window over the previously gathered sensor results. The maximum value of all sensor values is copied into the sliding window for the considered time slot. Third, we constitute the majority activity classification of all sliding window slots. In the last step, the maximum of all the sensor values and the sliding window determines the actual output value i.e. the current interaction activity classification.

This activity classification likely overestimates the user's interaction activity due to the majority voting within the sliding window. However, we argue in favour of a stabilised activity class as it has positive effects for our user interaction considerations. This rather cautious approach with an assumed user activity that is probably too high will only result in a different notification behaviour. If we were underestimating the interaction activity, the user would still feel disturbed by adaptations.

Figure 10.4 illustrates this four-step process in an example where the user starts an application to enter some text in reaction to information provided by the application. Two sensors determine the user's activity, an activity sensor that monitors running applications and a touch screen sensor that senses touch input. The sliding window has a length of 15 time slots, i.e. 15 seconds. At first, the application is started and both sensors detect low activity. For each time slot, the maximum value of both sensors is used for the respective time slot of the sliding window. For the first user activity this value is undetermined activity for all three time slots. Then, we determine the majority value of the sliding window which is also undetermined activity. The final output results from the maximum sensor output and the sliding window majority. In the following, the user continues to enter text via touch input, sends the text, and waits for the reply. He reads the reply and continues to use the application. With the example in Figure 10.4 we see a stable interaction activity classification using the presented mechanism.

The user interaction activity detection algorithm has been implemented using the MUSIC context model (cf. Section 8.2.1). Each activity detection sensor is implemented as a context sensor, which publishes its activity value using the MUSIC context ontology. The main activity classification mechanism, as depicted in Figure 10.4, is implemented as user activity context reasoner. This reasoner registers itself on the available activity sensors, i.e. the activity reasoner immediately receives updates from the activity sensors once they detect changes in the user activity. As the context reasoner is a special type of context plugin, its information can be used by other components of the middleware or applications. For example, the adaptation controller can evaluate the current user activity before initialising an adaptation or an adaptation reasoning mechanism can use the information on user activity during template evaluation. And last but not least, an application could interpret the current user activity and decide which type of notification to show (cf. Section 9.1).

We developed five different context sensors for activity detection on the Android operating system: ActivitySensor, PhoneStateSensor, ScreenSensor, TouchInputSensor, and TaskSensor. We will now describe each of the sensors more detailed:

**KeylockSensor.** The KeylockSensor monitors the status of the device's key lock. Users can configure a key lock that prevents unauthorised access to the device. A key...
lock can be a PIN code, a password, or a touch pattern. Removing the key lock is a clear indication of user activity. The Android OS sends system messages, so-called broadcast intents, to registered listeners whenever the key lock is removed or activated. The KeylockSensor registers as a listener and stores the moment when the key lock has been removed the last time.

PhoneStateSensor. The PhoneStateSensor monitors the telephone status of the device i.e. it detects whenever the user is making calls or getting called. The sensor can differentiate between incoming and outgoing calls. It registers to the Android internal TelephonyManager and is notified whenever changes to the telephone status occur. The PhoneStateSensor stores the current status of the TelephonyManager.

ScreenSensor. The ScreenSensor monitors the device’s display, i.e. whether it is turned on or off. Again, the Android OS sends broadcast intents whenever the display is turned on or off. The ScreenSensor stores the current status of the screen. In case the screen is turned on it will also remember the time since it has been turned on.

TouchInputSensor. The TouchInputSensor monitors the touch screen of a device and intercepts any contact with the screen. Intercepting touch screen inputs is a sensitive task; it can be compared to key loggers on traditional keyboard devices. Hence, operating systems like Android do not provide interfaces to retrieve information on the touch screen events of users. However, the touch screen is a very valuable source when it comes to user activity detection. For this reason, we developed the TouchInputService that allows the TouchInputSensor to retrieve touch screen events. The TouchInputService makes use of the Android internal WindowManager to modify the hierarchy of views that are currently displayed. It creates a new view that is always on top of the view stack. This view intercepts all touch screen events, records them, and passes them to the next view in the view stack. In this way, we get all touch screen events and the actual device use is not restricted. The TouchInputSensor stores all touch screen events with timestamps. Then, it calculates mean and variance for activity classification over the last 10 seconds.

TaskSensor. The TaskSensor monitors changes in applications and application usage. It detects when users switch between applications (multi-tasking), tasks, or views (activities). Therefore, the TaskSensor registers at the Android internal PackageManager and at the ActivityManager to receive updates on the above elements. The ActivityManager provides dynamic information on running processes, applications, and services. The PackageManager provides static information on applications, like their names or permissions. The TaskSensor combines dynamic with static information and stores the last moment in time for specified applications, tasks, and activities.

---

2 A view is the basic building block for creating user interface components. According to the Android API documentation, “a view occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for widgets, which are used to create interactive UI components (buttons, text fields, etc.).”.

3 Tasks describe collections of activities to maintain a seamless user experience within coherent user jobs. Whenever an application is started, it has at least one task that is also started.

118 Temporal User Participation
10.7 Summary

In this chapter we presented six mechanisms to achieve temporal user participation. Pre- and post-controllability are closely linked to the notification and feedback concept presented in Chapter 9. They allow ad-hoc influence of adaptations either right before an adaptation will be executed or right after it has been executed. In this way, a user can accept, postpone, reject, or undo adaptations. Furthermore, pre- and post-controllability is used during user focus reasoning to deviate from the focus-based decisions.

Manual adaptation stands in opposition to autonomous adaptation. With manual adaptation, users decide when they want the application to adapt. However, the underlying adaptation middleware must not be neglected and therefore we employ the concept of plan preferring to realise manual adaptation. The adaptation middleware considers the preferred realisation plans during adaptation reasoning. Moreover, manual adaptation is used when adaptive behaviour is disabled or when users want to deviate from user focus reasoning.

Starting and stopping adaptation allows users to disable adaptive behaviour temporarily or in the long term. While disabling adaptation temporarily can be configured per application, toggling the overall adaptive behaviour affects the entire adaptation middleware and therefore all applications.

User focus reasoning makes use of static information provided in the variability model of the application. User focuses should respect different tasks of the user within the application’s structure. With enabled user focus reasoning the adaptation middleware must not adapt application variants that contain a user focus which does not belong to the current configuration template.

Interaction activity detection is a solution for dynamic user focus reasoning. By determining the current interaction activity class we can decide whether to adapt or not. Furthermore, we can adjust the type of notification depending on the activity class. We provide a context plugin to determine and publish the activity information. This context plugin evaluates several other device sensors to determine the interaction activity of the user. However, interaction activity detection has not yet been integrated into the adaptation or notification decision of the middleware. The current implementation focuses on interaction activity related to the display, but there could be other types of interaction activity which would require new types of sensors, e.g. for the microphone or the camera.
11 Behavioural User Participation

This chapter covers the mechanisms from the behavioural participation dimension. This includes dynamic utility building with dynamic utility switching addressing utility function approaches as used for example in the MUSIC project. Further, we present a fuzzy-logic-based and a learning-based reasoning approach. Both are supposed to be more user-oriented than those employing utility functions.

Unlike the temporal participation mechanisms, the mechanisms for the behavioural dimension can only be used exclusively. That means, we can either use dynamic utility function building and switching, fuzzy-logic-based reasoning, or learning-based reasoning. The presented concepts make a few assumptions: first, we assume planning-based self-adaptation in a component-based software model with plans and plan properties. Second, we assume a single utility function per application; multiple functions as proposed by Khan [193] would make user participation more confusing. And third, adaptation is done according to changes in sensor information, i.e. application variants have a specific utility in each situation described by context information.

11.1 Dynamic Utility Functions

In Section 3.4 we already introduced utility functions as an adaptation reasoning policy for self-adaptive software and the problems they may raise. MUSIC implements a utility-function-based reasoning approach although it is entirely up to the application developer how the concrete utility function is constructed. In Section 8.2 we have already seen that this function does not need to be an actual mathematical function mapped onto the interval $[0, 1] \in \mathbb{R}$. Which sounds rather simple, turns out to be difficult in practice – especially with many application variants. A developer has to find a function that is optimised for each variant given the context and plan properties. Hence, we have a multi-objective optimisation problem. This is one of the reasons why MUSIC is so non-restrictive on the definition of utility functions. Because the resulting utility function is an individual compound developed only for one particular application, it cannot be reused in other contexts. The MDD process only supports the creation of utility functions by providing access to required context dependencies and plan properties. Currently, the utility function is the only part of the self-adaptive architecture that cannot be specified using modelling techniques. This means, only the original developer or a comparable domain experts can alter the adaptive behaviour defined by the utility function. Moreover, the utility function is specified during development-time and no changes are foreseen beyond this moment – to change the adaptive behaviour, the application must be re-compiled.
To achieve user participation in an adaptation approach that makes use of the utility function policy, we state the following four requirements:

**REQ1** Allow run-time changes of the utility function.

**REQ2** Simplify and systemise the development of utility functions.

**REQ3** Allow switching between different utility functions at run-time (profiles).

**REQ4** Make utility functions reusable for other applications.

To address **REQ1** and **REQ2** our solution exacts a consequent modularisation of one utility function into multiple sub-utility functions. Each sub-utility function determines the utility of a single quality metric. Hence, a sub-utility function reduces the complexity from a multi-objective to a single objective problem. Subsequently, we define the elements of dynamic utility functions.

**Definition plan property:** A plan property is a static value assigned to a single atomic or composite plan. It describes a specific property of a plan which is used by the single, non-dynamic utility function to assess the plan performance given a particular context. These properties are defined by the developer and are static at run-time. For example, the plan property `energy consumption` describes how much energy the component (described by the plan) would use when executed. If we measured the current energy consumption, we would specify this as a context dependency.

**Definition context dependency:** To adapt to changing context information, plans can specify context dependencies. Whenever one of the providing context sensors updates its information, re-adaptation is triggered and the new context value will be evaluated. A context dependency is specified using the MUSIC ontology (cf. Section 8.2.1). The context information can be represented as a primitive data type or a string value. For example, the context dependency `currentBatteryLoad` is provided by the `BatterySensor` and can be either represented as a percentage value relative to the maximum capacity or as a double value in Wh.

**Definition quality metric (QM):** A quality metric is a specialisation of a plan property, i.e. it has a constant value describing a plan. But unlike a plan property, it can be used in dynamic utility functions where it is evaluated by a sub-utility function. Therefore, it has to define the relation to one or more context dependencies, e.g. a quality metric `powerConsumption` relates to the context information `currentBatteryLoad`. This relation might be a mathematical function, too. In other words: the quality metric describes the power consumption of the plan (because it is a plan property) and for example, a plan with high power consumption should not have a high utility when the current battery load is low. Hence, the utility value of the `powerConsumption` metric is defined as the absolute difference of battery load and power consumption.

**Definition sub-utility function (SUF):** A sub-utility function calculates the utility of a single quality metric over a specified input domain. It uses one of the given primitive functions described in the next subsection. If a primitive function is not sufficient, a developer can add a custom function matching the requirements. A second parameter `inverted` specifies whether a high or a low input value determines a higher utility. Hence, a sub-utility function is defined as

\[
SUF := f(QM, inverted).
\]
**Definition dynamic utility function (DUF):** A dynamic utility function can be created and altered at run-time. It consists of one or more sub-utility functions (SUF), each calculating the sub-utility value of a quality metric. Weights for each of the sub-utility functions assess their importance in the overall function. The dynamic utility function determines the overall utility and makes sure it will return 1 as a maximum value:

\[
DUF := \max \left( 1, \sum_{i=1}^{n} w_i \times SUF_i \right); \quad \sum_{i=1}^{n} w_i = 1; \quad w_i \in [0, 1]\mathbb{R}.
\] (11.2)

### 11.1.1 Types of Utility Functions

Utility functions can be realised by arbitrary functions which makes it hard for developers to choose an adequate function that solves the given problem. It is even harder if the concept foresees only a single function per application as it is done in MUSIC. By the introduction of sub-utility functions, we reduce the complexity to a single quality metric with only a few input parameters (context dependencies and the constant plan property). This simplification allows us to give more advice and aid during development by providing five standard function types: step, linear, sigmoidal, piece-wise linear, and tabular. While the first four address a numeric input domain, the tabular function refers to a non-numeric input domain. Figure 11.1 illustrates the first four types of functions. The tabular utility function does not follow a classical mathematical functions scheme. All five types capture various patterns of utility value distribution and can be used without modification. If not sufficient, we still allow developers to define their own (static) utility functions or their own sub-utility function.

The simplest function is a **step utility function** which has only one parameter. The parameter separates the input domain into the two intervals acceptable (utility set to 1) and unacceptable (utility set to 0). The transition between the two intervals is a jump discontinuity, i.e. there are no intermediate values. A step utility function with one parameter \( y \) is formally defined as

\[
u(x; y) = \begin{cases} 0, & x \leq y, \\ 1, & x > y. \end{cases}
\] (11.3)

The **linear utility function** is described by two parameters. The input domain is now separated into the three intervals acceptable, unacceptable, and tolerable. The transition between acceptable and unacceptable follows a linear function and the utility value for tolerable will be somewhere between 0 and 1. A linear utility function with two parameters \( y_1 \) and \( y_2 \) is formally defined as

\[
u(x; y_1, y_2) = \begin{cases} 0, & x \leq y_1, \\ \frac{y - y_1}{y_2 - y_1}, & y_1 < x < y_2, \\ 1, & x \geq y_2. \end{cases}
\] (11.4)

Similar to the linear function is the **sigmoidal utility function** which is also specified by two parameters. The sigmoidal function uses are continuous transition from the unacceptable
to the tolerable and from the tolerable to the acceptable interval. However, the maximum
gradient is much higher than the gradient of the linear function. It addresses the more
natural case that transitions between two intervals are not abrupt. A sigmoidal utility
function with two parameters $y_1$ and $y_2$ is formally defined as

\[
u(x; y_1, y_2) = \begin{cases} 
0, & x \leq y_1, \\
1 / \left(1 + \exp \left(\frac{5(y_2+y_1-2x)}{y_2-y_1}\right)\right), & y_1 < x < y_2, \\
1, & x \geq y_2.
\end{cases}
\] (11.5)

A piece-wise linear utility function can capture more complex distributions of utility values
and is not limited to two intervals as with the standard linear utility function. We can
have many different utility entries for the input domain; each can be assigned a different
utility value. Transitions between two adjacent entries are linearly interpolated. A
piece-wise linear utility function is formally defined as

\[
u(x; y_1 \ldots y_n, z_1 \ldots z_n) = \begin{cases} 
z_1, & x \geq y_1, \\
z_i + \left(\frac{z_i+1-z_i}{y_{i+1}-y_i}\right) x, & y_i < x < y_{i+1}, \\
z_n, & x \geq y_n,
\end{cases}
\] (11.6)

whereas $(y_i, z_i)$ defines the $i$th utility entry and $n$ the number of entries.

A tabular utility function can be used for non-numeric input domains. It is specified by
an arbitrary number of parameters, i.e. utility intervals. Unlike the previous four types
of functions, it does not follow a specific mathematical function but rather it can be
seen as look-up table or a number of IF-THEN statements. For each (non-numeric) input
value the proper utility value is determined by looking it up in the table. A default utility
value is required whenever an input value is not explicitly specified. The tabular utility
function is similar to the piece-wise linear function, but for non-numeric input domains.
It is formally defined as

\[
u(x; y_1 \ldots y_n, z_1 \ldots z_n, z_d) = \begin{cases} 
z_i, & x = y_i, \\
z_d, & x \neq y_i \forall i,
\end{cases}
\] (11.7)

whereas $(y_i, z_i)$ defines the $i$th utility entry and $z_d$ the default value if no matching entry
has been found for an input value $x$.

11.1.2 Building Dynamic Utility Functions

We simplify the creation of utility functions by providing predefined sub-utility function
types that are used to assess the utility of single quality metrics according to current
context information. However, we have to know which quality metrics exist. This depends
on the application and its variability model. Therefore, we let the application developer
decide which quality metrics can be adjusted in the user interface of the middleware later
on. We call this additional information application meta data. These meta data can be
modelled using the MDD tools with the extended UML profile (cf. Section C.2). During
model transformation, an additional class ApplicationMetadata is added to an application
Figure 11.1: Basic types of utility functions. A high input value results in a high utility. This behaviour can be inverted by setting the *inverted* parameter of a sub-utility function.

bundle. Alternatively, a developer can create the required information manually. If the overall utility function should make use of the newly introduced quality metrics (instead of ordinary plan properties), application plans have to refer to the quality metrics provided by the *ApplicationMetadata* class. The developer has to initialise each of the quality metrics. Table 11.1 lists properties of a quality metric which have to be defined by the developer to achieve proper default behaviour. Some of the properties are variable and can be altered by the user via the middleware GUI at run-time. A dynamic utility function is completely described by its quality metrics and their meta data. Moreover, dynamic utility functions have a user-defined string identifier that separates one function from another and helps users to recognise their previously created functions.

The creation and modification of dynamic utility functions (based on the predefined types of sub-utility functions) is done from the middleware’s user interface. We provide two types of user interfaces, one using a Java Swing implementation for Windows and Linux and one for the Android operating system. Especially, the Android user interface shows how we can achieve utility function creation on size-constrained screens of mobile devices (cf. Figure 11.2). When creating a dynamic utility function, the user first selects the quality metrics he wants to include. He can select if the utility function should be
inverted, i.e. follows the less-is-better calculation scheme. In the second step, evaluation type and utility function type for this sub-utility function have to be specified. The evaluation type defines the relation between the context dependencies and the plan property.

Table 11.1: Specification of quality metrics. Except two optional properties, all remaining properties have to be specified by a developer as initial values. Some variable properties can be changed by the user later on.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the quality metric.</td>
</tr>
<tr>
<td>contextReference</td>
<td>String List</td>
<td>The context dependencies to which this metric refers. A context reference is a string as defined in the context ontology.</td>
</tr>
<tr>
<td>numericDomain</td>
<td>Boolean</td>
<td>Defines whether the input domain of the context reference is numeric or non-numeric.</td>
</tr>
<tr>
<td>minMax</td>
<td>MinMaxPair</td>
<td>Numeric domain: defines the lower and upper bound of the value range.</td>
</tr>
<tr>
<td>values</td>
<td>Object List</td>
<td>Non-numeric domain: defines the discrete values of the domain.</td>
</tr>
<tr>
<td>evalType</td>
<td>String</td>
<td>Defines how context dependencies and the plan property relate to each other. By now, we support proportional and inversely proportional relations. Custom evaluators can be added.</td>
</tr>
<tr>
<td>functionType</td>
<td>String</td>
<td>The type of sub-utility function used for this quality metric.</td>
</tr>
<tr>
<td>parameters</td>
<td>Object List</td>
<td>Whenever the sub-utility function requires parameters (most do), they have to be specified.</td>
</tr>
<tr>
<td>inverted</td>
<td>Boolean</td>
<td>The function types presented in Figure 11.1 use a more-is-better scheme, i.e. a high input value results in a high utility value. This parameter inverts the behaviour to a less-is-better scheme.</td>
</tr>
<tr>
<td>customEvaluator</td>
<td>ICustomEvaluator</td>
<td>If one of the predefined evaluation types is not sufficient, developers can specify their own types by implementing the ICustomEvaluator interface.</td>
</tr>
<tr>
<td>customFunction</td>
<td>ISubUtilityFunction</td>
<td>If one of the predefined sub-utility functions is not sufficient, developers can specify their own type of function by implementing the ISubUtilityFunction interface.</td>
</tr>
</tbody>
</table>
Currently, we provide two evaluation types, proportional and inversely proportional addressing the relation of a quality metric to a single context dependency. With proportional, the plan property and the context dependency have to match to yield a high utility. The inversely proportional type yields in a high utility when the difference between plan properties and context dependencies is higher. A developer can specify custom evaluation types by adding these to the application (using the ICustomEvaluator interface). A user can only select between the predefined evaluation types and those provided with the application itself. The last step is to choose the actual sub-utility function for this quality metric. Depending on the selected function, a user may have to specify additional parameters. Further, the user can specify a custom weight for this sub-utility function to define its importance compared to sub-utility functions of other quality metrics. If the field is left empty, an equally distributed weight is automatically chosen when building the DUF.

Because dynamic functions are created at run-time by users or developers, we have to store them persistently to make them available for later starts of the middleware and its applications. We developed a preferences manager for the adaptation middleware which manages all run-time preference settings by users. All preferences and hence dynamic utility functions are stored on a user and application basis. During the start of the middleware, user preferences are loaded and initialised by the preferences manager.

11.1.3 Evaluating Dynamic Utility Functions

The evaluation procedure of normal utility functions within the MUSIC middleware is simple. The adaptation reasoning algorithm (e.g. the brute force reasoner) calls the evaluate method of the application’s utility function. The developer has to make sure
that this method returns a valid utility value within $[0, 1]$. We will call this type of utility function \textit{static utility function} as they are defined at design-time and cannot be changed afterwards.

\begin{algorithm}
\begin{algorithmic}
\ Function \ \ \textsc{evaluate}
\ For all qualityMetrics do
\ \ \ \ \ If metric.weight > 0 AND metric.isEnabled then
\ \ \ \ \ \ \ \ customWeight = true;
\ \ \ \ \ \ \ \ break
\ \ \ \ \ Else if metric.isEnabled then
\ \ \ \ \ \ \ \ numEnabled ← numEnabled + numEnabled
\ \ \ \ \ End if
\ End for
\ For all qualityMetrics do
\ \ \ \ \ If metric.isEnabled == false then
\ \ \ \ \ \ \ \ continue
\ \ \ \ \ End if
\ \ \ \ \ If metric.hasCustomEvaluator() then
\ \ \ \ \ \ \ \ evalValue ← customEvaluator.calculateEval()
\ \ \ \ \ Else
\ \ \ \ \ \ \ \ contextReference ← metric.reference
\ \ \ \ \ \ \ \ contextValue ← getContextValue(contextReference)
\ \ \ \ \ \ \ \ planPropertyValue ← getPropertyValue(contextReference)
\ \ \ \ \ \ \ \ evalType ← metric.evalType
\ \ \ \ \ \ \ \ evalValue ← calculate(contextValue, planPropertyValue, evalType)
\ \ \ \ \ End if
\ \ \ \ \ If metric.hasCustomUtilityFunction then
\ \ \ \ \ \ \ \ metric.utility ← metric.customUtility.evaluate()
\ \ \ \ \ Else
\ \ \ \ \ \ \ \ metric.utility ← metric.subFunction.evaluate()
\ \ \ \ \ End if
\ \ \ \ \ If customWeight then
\ \ \ \ \ \ \ \ weight ← metric.weight
\ \ \ \ \ Else
\ \ \ \ \ \ \ \ weight ← 1/numEnabled
\ \ \ \ \ End if
\ \ \ \ \ metric.utility ← metric.utility * weight
\ \ \ \ \ utility ← utility + metric.utility
\ End for
\ Return utility
\ End function
\end{algorithmic}
\end{algorithm}

With dynamic utility functions, we moved the implementation and evaluation complexity of utility functions into the middleware. The \textit{DynamicUtilityFunction} class extends the abstract class \textit{AbstractPropertyEvaluator} of the MUSIC middleware like static utility functions do. Hence, it integrates transparently into the adaptation process – the
adaptation reasoning algorithm does not know if the evaluating function is static or dynamic. Algorithm 11.1 lists the evaluation procedure for DUFs. At first, we check for all quality metrics associated with the DUF if the metric is enabled and if it has a custom weight set. If yes, we will use custom weights when determining the overall utility value of the DUF. If not, we must count all enabled quality metrics to use an equal weight distribution when determining the overall utility value in the end. In the next step, the actual domain input for the utility function is determined by calculating the evaluation value which determines the relation between context dependencies and the constant plan property of the quality metric. The sub-utility function takes the evaluation value and calculates the utility for this particular quality metric. If set, custom evaluators or custom sub-utility functions are used. Once all utility values are calculated, the evaluation process for the DUF returns the overall utility value as a weighted sum of each of the quality metric utility values. The evaluation process is now completed.

11.1.4 Switching Dynamic and Static Utility Functions

Each utility function represents a specific application behaviour. That means, each function is a behaviour profile and could represent the individual requirements for different users. Such profiles comply with ISO 9241-129:2010 [68] and the request for setting profiles by Kniwel et al. [223]. While static utility functions define (stable) default behaviour of the application, dynamic utility functions let users alter this behaviour at run-time if the default behaviour is not appropriate. Different profiles allow users to experiment with dynamic utility function without doing any damage. In case of a malfunctioning utility function, a user can simply switch back to an old profile. In this subsection will we show how to define multiple utility functions and how we can switch between different utility functions, whether they are static or dynamic.

We have seen that self-adaptive applications do not foresee more than one configuration for different use cases or users. Also, MUSIC lets developers define only a single utility function that has do determine the application behaviour in every possible case. This makes the application behaviour either very limited to a special use case or the engineering process for the utility function becomes very difficult as many possible cases have to be covered. However, in both cases, the utility function is not flexible enough.

We extend this concept by letting developers define multiple static utility functions during application development to consider different use cases and users from the very beginning. These multiple utility functions are grouped into so-called static utility function containers. If required, developers can model multiple static utility functions with the extended UML modelling profile. One of the utility functions has to be marked as default so that the adaptation middleware knows which function to evaluate if a user has not set his preference yet. When multiple functions are present, the model transformation process automatically creates a utility function container implementing the IStaticUtilityFunctionContainer interface.

Dynamic utility functions are created at run-time and are grouped into dynamic utility function containers. Each self-adaptive application has a single dynamic utility function container which is created alongside with the first DUF. From the user interface, the desired function can be selected as the currently active function (cf. Figure 11.2).
During adaptation reasoning, the correct utility function has to be selected. Hence, we first check whether the application contains a static utility function container or any user-defined dynamic utility functions within a dynamic utility function container. If yes, we query the preferences manager for the currently selected utility function. If the application has a utility function container, but no active utility function, the default (static) function will be returned. The adaptation reasoning algorithms finally evaluates the returned utility function.

11.1.5 Summary

The presented concept of creating and switching dynamic utility functions allows flexible run-time management of utility functions and is the foundation for user profiles. Furthermore, developers can provide multiple static utility functions to allow flexible application behaviour from the very beginning. For our MUSIC implementation we allow downward compatibility to applications that do not support multiple and dynamic utility functions. Moving the building process for dynamic utility functions into the middleware allows a clear separation of concerns. The user interface to alter utility functions is located in the middleware, too, i.e. separated from the application logic. In case the user misconfigures the utility function, he is still able to undo his changes as the middleware itself is not affected. Although the user can approach the perfect utility function by trial and error, this is one of the main problems as inappropriate utility functions can result in unstable application behaviour. There are several more limitations: first, utility functions are still difficult to understand and to set-up. Dynamic utility functions simplify the management of such function, but the problem remains; only expert users or developers will benefit from DUFs. Second, the developer must prepare the usage of DUFs properly. If he does not provide the necessary quality metrics with its properties, DUFs cannot be used. And third, a quality metric represents one constant plan property that relates to one or more context dependencies. We leave support for multiple plan properties and predefined evaluation types for multiple context dependencies for future work.

11.2 Fuzzy-Logic-Based Adaptation Reasoning

The set-up, alteration, and management of utility functions is a cumbersome task and requires deep knowledge on the application’s variability capabilities, available context information, and the mathematical foundations. Unlike expert users and developers, ordinary users are probably not able to modify the application behaviour in this way. Chuang and Chan [184] add that “typical mobile application users are not concerned about the low-level resource parameters or the contextual factors. They are concerned with what they can perceive about the performance of the application in terms of, for example, responsiveness, smoothness, and clarity” [184]. Chuang and Chan define concrete user-oriented measurable QoS parameters similar to our quality metrics used by dynamic utility functions. They suggest making these parameters easier to understand and to modify by ordinary users. We pick up the idea and integrate the quality metrics within a fuzzy-logic-based adaptation reasoning approach. Fuzzy logic promises stable and user-understandable adaptation reasoning. For more details on fuzzy logic and control see Chapter 6.
An adaptation reasoning approach based on fuzzy logic should fulfil the following four requirements:

**REQ1** Allow easier run-time modification of the application behaviour.

**REQ2** Allow switching between different previously defined behaviour profiles.

**REQ3** Achieve robust adaptation reasoning even on erroneous user input.

**REQ4** Improve intelligibility and mental model correlation of users.

Although working on a rule base, fuzzy control is similar to utility-function-based reasoning. The membership domain of fuzzy variables is in the interval $[0, 1]$ as well.

Based on the MUSIC adaptation model we suggest two different approaches to include fuzzy control into a component-based self-adaptation middleware:

1. **Straightforward approach:** The current performance (i.e. utility) of an application variant is directly determined by the rule base. Context dependencies and plan properties are antecedents resulting in the concrete utility of the application variant. For example: IF $BatteryLoad$ IS high AND $PowerConsumption$ IS low THEN $Utility$ IS high. The natural understanding of this rule would be something like: “if my device’s battery load is still high and the component’s power consumption is low, then this plan variant is preferred”. With this alternative, the user has to know about all application variants, their plan properties, and the structural composition of the application. Moreover, each rule has to be plan-specific, because it addresses specific plan properties.

2. **User-oriented approach:** This alternative can be seen as a hybrid way between utility function and fuzzy reasoning. In the first step, a developer sets-up the utility function. But instead of comparing constant plan properties with context dependencies, the developer creates so-called fuzzy metrics and compares them with the constant plan properties to achieve the overall utility of an application variant. During the second step, the user creates rules to determine the outcome of the fuzzy metrics. This time, only the context dependencies work as antecedents and the user can select which available context information yields a particular fuzzy metric. For example: IF $BatteryLoad$ IS high THEN $FuzzyPowerConsumption$ IS high; whereas $FuzzyPowerConsumption$ specifies the fuzzy metric. The natural understanding of this would be something like: “if my device’s battery load is still high, the power consumption of the application should be high, too”. Then, it is up to the system to achieve high power consumption for the application.

Even though the straightforward approach promises to be more easily to implement and understandable for developers and experts in the field of fuzzy control, the user-oriented approach is assumed to be better understandable by ordinary end-users. The person who creates the rule base does not need knowledge on internal information of the application, but rather rules can be created based on common sense. Creating rules from context dependencies to define a metric can be done without expert knowledge. At the same time we do not lose the power of the utility function approach that scales well with the number of application variants. However, the drawback here is the definition of the developer-defined utility function. Although we allow users to modify the run-time behaviour, the performance of the application stands and falls with a proper definition of the underlying utility function.
Figure 11.3 illustrates the user-oriented fuzzy reasoning approach, which we have implemented for this work. The fuzzy reasoning component takes context information as reference (input). The output determines the value of a fuzzy metric. The fuzzification of the context information and the fuzzy metrics is provided via the application metadata. Applications define only dependencies on the fuzzy metrics and re-adaptation is triggered once the fuzzy metrics change.

Fuzzy reasoning involves the three steps fuzzification, inference, and defuzzification. In the following, we will describe this procedure in more detail, as well as the creation of fuzzy profiles each representing one rule base. Of course, all previous considerations and following details are specific to the underlying MUSIC approach, but they are exemplary for the application of fuzzy control to self-adaptive software.

11.2.1 Fuzzy Profiles

We already introduced behaviour profiles with dynamic utility functions. To comply with REQ2 we do the same for the fuzzy-logic-based reasoning. A fuzzy profile defines a rule base for a particular application. Users can create as many fuzzy profiles for an application as they want to reflect their individual needs.

Users can define rules via a user interface of the middleware to achieve a separation of concerns. Even with misconfigured fuzzy profiles, the user is able to repair the application.
behaviour or to restore an old profile via the middleware’s user interface. Each rule is built from fuzzy variables and its linguistic terms. A variable can be either an input (antecedent) or an output (consequence) variable. Following the user-oriented approach, the input variables are context dependencies and the output variables are fuzzy metrics. For the straightforward approach, the input variables would be context dependencies and plan properties while the output variable would be the utility.

Context dependencies, fuzzy metrics, and their meta data are defined in the application bundle. When the bundle is installed or new plans are registered, the meta data is stored on an application-basis in the preferences manager (introduced with dynamic utility functions). Context dependencies and fuzzy metrics are then available in the user interface for creating fuzzy rules. This means, only context dependencies and fuzzy metric defined within the application meta data can be used for the creation of rules.

Figure 11.4 illustrates the middleware user interface for the management of fuzzy profiles on the Android system. Users can create, delete, or rename entire fuzzy profiles. One of the available profiles can be set as active. For each of the profiles, any number of rules can be defined. At the moment, we only support AND conjugation of fuzzy variables.

![Profile management. Creating a fuzzy profile. Specifying a rule.](image)

**Figure 11.4:** Middleware user interface for managing fuzzy profiles on the Android system.

11.2.2 Fuzzification

For the fuzzification process, membership functions must be assigned to every linguistic term of a fuzzy variable. Following the user-oriented approach, we have two types of fuzzy variables – context dependencies and fuzzy metrics. Before assigning the membership functions, the linguistic terms for each of the variables must be defined. In Chapter 6 we have seen FCL descriptions which define all necessary information. However, providing FCL descriptions requires additional file parsing (and hence time.
and computational overhead) and causes a conceptual break in the MDD process. Thus, we decided to model linguistic terms during the MDD process in a FCL-compatible way. Alternatively, a developer can specify the necessary information manually in the application bundle.

Figure 11.5: Fuzzification of context dependencies and fuzzy metrics: association of linguistic terms ($mFuzzyTerm$) to the fuzzy variables $BatteryLoad$ and $PowerConsumption$ (both $mPropertyType$) for the HelloWorldApp (cf. Section D.2), modelled with the extended UML profile (cf. Section C.2).

Figure 11.5 depicts the two fuzzy variables $BatteryLoad$ and $FuzzyPowerConsumption$. The linguistic terms are modelled as $mFuzzyTerm$ and define which type of membership function is used as well as the required function parameters. Both, context dependencies and fuzzy metrics are modelled as $mPropertyType$. It depends on their assignment within the utility function how they are actually transformed into program code.

To achieve downward compatibility with existing applications, we use the same concept as with dynamic utility functions and define application meta data. All fuzzy-control-related information is specified within the optional $ApplicationMetadata$ class. When using the MDD process, all required information is generated automatically during the transformation process.

For the fuzzy metrics we use the same approach as with quality metrics and attach the meta data to the metric, but we have to remember that fuzzy metrics are actually context information, while quality metrics define constant-valued plan properties. This might seem awkward in the first moment because it means to assign fuzzy metric meta data to plan properties. However, when thinking of the still used utility function this may become clearer: a developer has to compare one or more fuzzy metrics with one or more constant properties of a plan. Hence, it is reasonable to attach the existence of a fuzzy metric to a plan property. For example, the fuzzy metric $FuzzyPowerConsumption$ is attached to the constant plan property $powerConsumption$ defining the power consumption of a plan. In
contrast, the fuzzy metric is used to express when power consumption is supposed to be high, medium, or low (based on predefined context information). Within in the utility function, the fuzzy metric and the plan property are compared and the overall utility is determined.

Context dependencies of plans were only defined as string values with a reference to the context ontology (cf. Section 8.2.1). Adding meta data like linguistic terms is not possible in this way. Hence, we created a new class ContextDependency that can hold additional ContextDependencyMetadata. Again, for reasons of downward compatibility, we always keep the old definition of context dependencies and make the new definition optional respectively additional. Context dependencies which are defined as fuzzy variables will not cause an application to re-adapt if their values changed. Instead the fuzzy reasoner is triggered for re-evaluation.

### 11.2.3 Inference and Defuzzification

With the definition of fuzzy variables and their fuzzification, we can do inference and determine the sharp output values for the fuzzy metrics. This process is initiated whenever one of the following conditions is met:

1. The user sets another fuzzy profile as the active profile.
2. The rule base of the active profile has been changed, i.e. rules were added, removed, or altered.
3. The context information, required by the rule base of the active profile, has changed.

When a new fuzzy profile is set, several operations are necessary before the reasoning can begin. At first, we check whether the new profile is different to the current profile. If yes, the context listeners of the current profile are removed and the context listeners for the new profile are registered. The fuzzy reasoner registers listeners on all context dependencies defined as antecedents. In case of context changes, the fuzzy reasoner will be notified to initiate a new reasoning process.

Before we start the reasoning process, we have to update all context dependencies required for the rules of the active profile, i.e. we have to get the latest values via the IContextAccess interface from the context middleware. For the actual inference and defuzzification process we use the fuzzy logic library jFuzzyLogic [29]. This library provides the required data structures and is fully FCL-compatible, especially for the creation of rules and their elements. Due to its open source license we were able to integrate its source code directly into the self-adaptation middleware without generating additional overhead by external libraries.

After defuzzification we have to publish the fuzzy metrics so that they can be used by an application’s utility function. We developed the Fuzzy Preferences Context Plugin for publishing the fuzzy values of the fuzzy metrics to the context middleware. Applications that make use of those fuzzy metrics will be notified on the update and the adaptation controller will initiate evaluation of the utility function.
11.2.4 Summary

In this section we showed how to apply fuzzy-based reasoning to component-based self-adaptation. We decided to use a hybrid reasoning approach with an underlying utility function in favour of understandability, intelligibility, and lower demands on users’ knowledge on self-adaptive behaviour. Using the hybrid solution allows us to combine the fuzzy reasoning solution with the dynamic utility function approach presented in Section 11.1. Developers can create a dynamic utility function that is based on fuzzy metrics. Thus, we achieve even more flexibility on run-time user participation; not only can we change the influence of context dependencies on the adaptive behaviour in a user-oriented way (fuzzy) but also the exact utility of particular application variants (dynamic utility function).

As with dynamic utility functions, the preferences manager plays a central role. It stores all fuzzy profile information for multiple users and multiple applications. Further, it initiates the fuzzy reasoning process when changes in the profiles were made. For the fuzzy reasoning process, context dependencies had to be enriched with additional meta data. Fuzzy metrics are defined in combination with plan properties that are used by developers within the utility function. A user interface design implementation for the Android operating system demonstrates fuzzy rule management on small screens.

11.3 Learning-Based Adaptation Reasoning

As the third solution for user participation in the behavioural dimension, we present a learning-based reasoning algorithm. Although dynamic utility functions and fuzzy-based reasoning allow modifications by the user, it is more desirable to have an application that behaves perfectly for an individual user. That means, instead of being adjusted by the user, the application adjusts itself to changing user requirements.

Speaking of learning typically involves machine learning as a sub-field of artificial intelligence. Machine learning techniques aim at automatically acquiring knowledge from data. Hence, we first need to define what type of data we have and what type of knowledge we are able to acquire. Given a component-based self-adaptive middleware as described in Chapter 8, variability is defined over the distinct application variants through the reasoning policy. There is a high degree of user satisfaction when the system fulfils the user’s requirements [161]. Consequently, selecting an application variant that meets the user’s expectation will result in a high user satisfaction. This assumes that there is an application variant that meets these expectations and we can react to changing user requirements accordingly.

In the remainder of this section we will briefly discuss what types of learning methods can be used to achieve learning in self-adaptive applications and subsequently which specific learning method we employ in our approach. Following, we present the way how it is integrated into the self-adaptive middleware and how users can interact with it.
11.3.1 Learning Method

Several machine learning algorithms are available and suitable from which we focus on supervised learning techniques only. That means we consider the classification of objects or the learning of functions based on labelled training data. Unsupervised or semi-supervised techniques are not taken into account as we always have the possibility of labelled training data in our use case. We will have a look at reinforcement learning, though. Or focus is not on the comparison of different techniques, but rather we are looking for one possible solution that meets our requirements. Hence, we define the following requirements that a learning technique for self-adaptive software should fulfil:

REQ1 It should be transparent and intelligible, so that users could optionally get explanations why something was learned or decided.

REQ2 It should allow run-time adjustments, i.e. have a dynamic learning model to reflect changes in the user's environment or habits.

REQ3 It should be able to generalise with little or no domain-specific initial knowledge.

REQ4 The effort to train the system should be kept to a minimum.

REQ5 The system has to generalise only for a single user.

Before considering which method to choose for learning-based adaptation reasoning, we need to define what do we want to learn. With a utility-function-based adaptation reasoning it is self-evident to learn the parameters of the function. This is a typical field for function approximation which can be done by neural networks, linear regression, or Gaussian processes. However, in the original MUSIC case we have only a single utility function per application. Depending on the number of application variants and plans, the number of parameters can be very high and most of the given methods will hardly generalise – especially with no or little initial domain knowledge. Learning parameters of the utility function is not very intelligible, too. It will not be possible to create any explanations from parameter changes.

Moving away from the utility function policy and switching to an action-based policy, i.e. a rule-based approach, gives us the opportunity to perform rule-based learning. Rule-based learning is often combined with decision trees or predicate logic systems. We decided against rule-based learning as either developers or users have to create a complex set of rules, i.e. to provide a large amount of initial knowledge. Especially users are likely to be overextended with this task.

Generally, when considering what can be actually learned, we have application variants on the one hand and user situations described by context information (that represent changing requirements) on the other hand. Learning situations means we provide a fixed set of application variants to the user who has to choose the situation that matches this variant best. However, learning application variants on the contrary would mean to provide a fixed set of situations and the user has to decide which application variant matches this situation best. Of course, one can learn both at the same time, but this would make the learning process complex and protracted.

We decided in favour of learning application variants for which several classification and learning techniques come into consideration. At first we will look at reinforcement learning (RL). RL does only need little or no initial domain knowledge and no explicit
system model. It has a more agent-oriented view with software agents executing particular actions and receiving rewards for their actions. In each moment, the agent can select between different actions. In our case such actions would be the different application variants. The goal of the agent is to get as much reward as possible. RL has its focus on on-line performance with finding a balance between exploration and exploitation. Learning application variants does not require good on-line performance and exploration. Using RL, we must also provide information on the possible rewards, which can be only done by the user properly. Generally speaking, RL would likely distract the user from its current task and we rather propose an algorithm that involves less or only implicit user interaction while learning.

The classic k-nearest neighbour (k-NN) classifier is a non-parametric function that uses the \( k \)-closest training examples as input, whereas \( k \) is a small positive integer. Its output is a class membership determined by a majority voting of the nearest neighbours. The evaluated instance is assigned to the class most common among its neighbours. Typically a distance metric like the Euclidean distance is used to assess which neighbours are taken into account.

The k-NN classifier belongs to the class of instance-based learners. Such classifiers do not perform explicit generalisation, but instead compare new problem instances with instances seen during the training phase. All trained problem instances have to be stored in some kind of knowledge base which is one of the disadvantages of instance-based classifiers. The entire dataset has to be kept in memory at run-time and may also involve a longer period of time when evaluating new instances with the stored dataset. One important advantage of instance-based classifiers is its ability to adapt its model to previously unseen instances. Other classifiers would require a new training run on the data. Further, the k-NN classifier is well understandable. The Euclidean distance can be used easily to generate explanations to the user. The algorithm itself is straightforward to implement and stable during execution. Thus, we decided to employ the k-NN classifier to classify application variants in relation to situations.

11.3.2 System Overview

For our learning-based adaptation reasoning we provide a two stage approach to ease intelligibility and traceability for users. Learning application variants allows us to let the user train the system during normal application usage which would not be possible otherwise. In the first stage we provide an abstraction from concrete context sensors into abstract situations. This has several advantages: we reduce the input space for the classification algorithm significantly and reduce chances of overfitting or no fitting at all. Assuming that there are fewer situations than sensor information defined, the learning performance will be improved. Moreover, the trained data is not dependent on concrete sensor implementations and information. It is possible to replace sensor instances, but keep the training data as it is based on the abstract situations. In the second stage, we perform the actual classification based on the user's training data. That means, we compare the current situation with the training data and select the application variant that would fit best according to the trained data. Hence, we use 1-nearest neighbour classifier with the Euclidean distance as distance metric between the current situation and the sampled data. For more details on the classification process we refer to Section 11.3.5 and Section 11.3.6.
Figure 11.6: Two-stage process for learning application variants. Learning of application variants is based on situations in the second stage. The situations are user-defined based on sensor information in the first stage. A fuzzy inference process derives situations from sensor data.

Figure 11.6 illustrates the overall approach. For the first stage (fuzzy inference) we employ a similar fuzzy reasoning mechanism as described in Section 11.2 to specify which context parameters define a situation. In this example, ten sensors provide information for eight situations. The battery sensor is not used at all and for the shopping situation no sensor has been assigned. In Section 11.3.3, we describe how situations are defined in detail. In the second stage (k-NN classification) the user selects the best fitting application variant during training mode. In operational mode, the system has to choose the best fitting application variant based on the training data. The system needs to determine the current situation description and compares it to the available situations.

11.3.3 Specifying Situations

Situations are derived during the first stage of the learning process. In this context, a situation $s_x$ is defined by a set of context parameters $c$:

$$s_x := \{c_{x1}, c_{x2}, \ldots, c_{xn}\}.$$  

(11.8)
A situation $s_i$ is different to a situation $s_j$ if they differ in at least one of their context parameters:

$$s_i \cap s_j \neq 0 \text{ if } \exists c_{ik} \neq c_{jk}; 0 < k < n.$$  \hspace{1cm} (11.9)

Each situation has a sharp scalar value within the interval that is determined during a fuzzy inference process:

$$|s_x| \in [0, 1] \mathbb{R}.$$  \hspace{1cm} (11.10)

Hence, context parameters and situations are fuzzy variables with linguistic terms and associated fuzzy set membership functions (cf. Chapter 6). Unlike with fuzzy reasoning (cf. Section 11.2), the context parameters are not limited to the context dependencies specified by the particular application. Instead, situations can be specified over all available context information, i.e. context information provided by all active context sensors. A user can precisely define which context information define a situation and does not take information into account that will not describe a situation sufficiently, e.g. `Brightness` to describe whether the user is working or not.

![Figure 11.7: Middleware user interfaces for managing situations on Android.](image)

Each fuzzy variable has to be fuzzified before inferring over the rule base. In Section 11.2, we simply provide the required fuzzification information as meta data along with the application bundle. However, when using all available and application-independent context information, the fuzzification must be done elsewhere. Therefore, we extended the `IPluginMetadata` interface of the context middleware with a method to get the linguistic terms for a particular context parameter, i.e. an entity-scope-pair. Thus, context plugin developers are forced to add fuzzification information to their plugins (which
may address multiple entity-scope-pairs). We think leaving the fuzzification process to the developers is reasonable. A plugin developer knows best what the context plugin provides, i.e. which type of information and also its value range. When a context plugin with linguistic terms is registered at the context manager, it stores the association of entity-scope-pairs and the linguistic terms. This allows us to retrieve linguistic terms for every entity-scope-pair via the IContextManagement interface. When a context plugin is removed from the system, the associated linguistic terms are removed, too.

The situations of the fuzzy inference system must be specified before starting with the learning process. Developers or ordinary application users specify situations once the application has been deployed on the middleware. Figure 11.7 depicts the user interfaces for the specification of situations for the Android operating system. Users can create new situations and specify afterwards which context parameters determine the situation by creating the respective fuzzy rules. The fuzzification of situations is more straightforward. We define three linguistic terms for a fuzzy situation variable: low, medium, and high. Each of the terms have a triangular membership function; all equally distributed in the domain \([0, 1]\). That means, defuzzifying a situation variable will always result in a sharp scalar value within \([0, 1]\). The linguistic terms for situations are dynamically created at run-time whenever a user adds a new situation to the rule base. It might be wise to encourage developers to provide few basic situations so that users do not need necessarily specify their own situations from the very beginning.

### 11.3.4 Identifying Application Variants

Application variants are represented by so-called configuration template hierarchies with one (child) configuration template per plan. The adaptation middleware builds the template hierarchy from the application’s variability model right before the adaptation reasoning step. We need this dynamic build process to reflect changes in the available plans. A configuration template hierarchy is a non-persistent, loose collection of plans describing components. The variability model does not model particular application variants, but instead variation points are implicitly defined. As a consequence of this rather loosely coupled structure, application variants cannot be addressed explicitly, i.e. there is no identifier describing a specific application variant. However, the proposed learning concept is based on the learning of application variants because users can grasp the concept of application variants more easily than a loosely collection of plans. During training, we have to store the variants selected by the user in a knowledge base to select the best template during operation later on. Therefore, we developed a fast identification mechanism to uniquely identify template hierarchies at run-time.

A template hierarchy is uniquely described by its plans and their parameter settings (in case of parametric adaptation). There are no two template hierarchies with the same plans and the same parameter settings. A unique identifier should include plan and parameter setting information, but we do not know beforehand how many and which plans will be in this particular template hierarchy in the end. Therefore, we recursively generate the plan identifier during creation of the template hierarchy. A configuration template hierarchy is generated whenever a template iterator (iterates over a single template hierarchy) is traversed. Thus, every time a configuration template for an atomic plan is created, the identifier is generated, too. Atomic plans are the terminal nodes of
the template hierarchy tree. The identifier is built from the plan’s name and associated parameter settings – both identify a plan uniquely. In case a configuration template is created for a composition plan, first the identifiers of all children are concatenated and then combined with the identifier of the composition plan. Finally, the root template includes a concatenated string of identifiers from all child templates.

Depending on the size of the template hierarchy, the string can be rather long. Although this string is human-readable and helps developers during debugging, it could cause problems besides its obvious lack of efficiency during the learning process. Thus, we make use of the Java hashcode function which reduces the length of the identifier to a 32 Bit Integer value.

11.3.5 Sampling and Training

The nearest neighbour classification algorithm requires a knowledge base with initial data to be trained before classifying situations during operational mode. The application-specific knowledge base is trained using an application without any autonomous adaptations at all. During training, the adaptation middleware does not initiate adaptation reasoning. Instead, a user works with the application as it would not support adaptations or adaptation reasoning is disabled (cf. manual adaptation in Section 10.3). This means, the application developer has to provide interfaces to access or enable all possible application variants from the user interface.

With manual adaptation the feedback manager takes feedback from the user regarding the preferred plan(s). When the user prefers a particular plan (or multiple plans), the middleware must use this input to update its knowledge base. We developed a Learning Manager as an OSGi component that encapsulates all training-related functionality. Hence, the learning manager has to register as a feedback listener to the feedback manager. This process is called situation sampling, because we sample the values of the current situation variables into a situation vector. The situation vector declares how much the current situation belongs to one of the predefined situations. It describes the current situation sufficiently and will allow us to classify future situations and select the right application variant. The situation vector is defined as

$$\vec{s} = (s_1, s_2, s_3, \ldots, s_n); s_i \in [0, 1]R.$$  \hspace{1cm} (11.11)

The structure of the knowledge base is shown in Equation 11.12. Each column of the matrix represents a specified situation and each row an application variant and therefore one situation vector as defined in Equation 11.11. With each sample procedure, we add the new situation vector to the knowledge base. If the sampled application variant does not exist yet, we append a new row to the matrix. Otherwise we add the situation vector to the existing entry. To determine an average situation vector for an application variant during reasoning, we have to remember how often we sampled a situation vector for a variant. This this is done via the count vector (cf. Equation 11.13). We will see an example of the training and classification process in Section 11.3.7.
Depending on the mode (training vs. operational), the modification of the knowledge base is different. The classic approach during training works as follows (after a user did a manual adaptation):

1. Get all applications that employ the preferred plan(s).
2. Get the root configuration template (for each application) which contains the preferred plan(s).
3. Create a situation sample: get the current scalar value for each fuzzy situation variable and store it as a situation vector.
4. Add the current situation vector to the knowledge base with the identifier of the root configuration template as key. In case there already exists an entry for this root template, add the new vector to the existing vector and increment the counter for this entry by one.

During operational mode, user feedback (i.e. manual adaptation) indicates a user's dissatisfaction with the autonomously selected application variant (based on the learned samples). That means, we have to adjust our knowledge base to better reflect the user needs. This is done by an adjustment weight $w \in \mathbb{R}$ which has to be specified by the application maintainer. A weight of $w = 1$ samples the situations as in training mode. A weight of $w = 0$ ignores the plan preference and any weight $w > 1$ progressively adjusts the knowledge base, i.e. the dissatisfaction of users is rated higher by the factor $w$ as situation samples during training mode. A weight $0 < w < 1$ degressively adjusts the knowledge base, i.e. the dissatisfaction of users has underproportional influence on the learning process.

### 11.3.6 Classifying Situations

In the following, we present a new adaptation reasoning algorithm that implements a k-nearest neighbour classification. Unlike all previous presented reasoning algorithms, the learning-based algorithm does not employ utility-function-based reasoning. However, we still use the term utility to describe a configurations template’s usefulness according to the current situation. The learning-based reasoning algorithm returns the best configuration template (hierarchy) for the current situation according to the k-nearest neighbour classification. Unlike classic self-adaptive reasoning where every reasoning process evaluates only the current context values, the result of the learning-based reasoning is solely based on previous observations, i.e. samples.
Before we determine the nearest neighbour, the current situation vector \( \vec{s}_{\text{cur}} \) has to be updated with the latest values. This is the vector for which we have to find the best matching configuration template hierarchy in the knowledge base. Subsequently, we look-up every available template hierarchy in the knowledge base. If a template hierarchy was not sampled during training mode, we will not find it in the knowledge base. This can be the case if the user did not select the application variant during training or if the application was created after training mode was completed. In such a case we ignore the template and it cannot be selected for the best matching template. In the worst case, no template is found in the knowledge base or the knowledge base is empty. In this case we make use of default template reasoning as introduced in Section 10.4. However, in most cases the template will be found and it is classified according to the current situation. At first, the situation vector from the knowledge base is divided by its sample count. In the next step, the average template vector is subtracted from the current situation vector as distance. These steps are repeated for all templates found in the knowledge base. The template with the smallest distance will be returned by the reasoning algorithm.

While this is the standard procedure for the nearest neighbour classification, we optimised the algorithm in regard to our fuzzy situation variables. The value of a situation variable is within the interval \([0, 1]\) and declares how much the current situation belongs to one of the predefined situations. If the average situation vector of a template contains a value with a membership around 0.5, then we cannot make a proper prediction for this situation. For the calculation of the nearest neighbour, this situation has the same weight as other situations that would characterise the current situation much better, e.g. situations around 0 or 1. Therefore, we use an additional parabolic weighing function that weighs situations with a membership around 0.5 as high and membership around 0 and 1 as low (cf. Figure 11.8). In the context of the k-NN algorithm, a higher value is worse than a lower value as it denotes a higher distance.

![Figure 11.8: Parabolic function to weigh a situation value \(|S_x|\). Situation values around 0.5 are less meaningful for the classification than values around 0 or 1. Therefore, we increase the distance (less suitable) for k-NN by a higher weighing factor.](image)

The algorithm for operational mode is listed in Algorithm 11.2 on page 146. It is a two-step process due to the required reasoning over multiple parallel applications. First, it performs the knowledge base classification to find the best suitable template. However, if no template was found, a second run with default template reasoning is done for the
application for which no entries in the knowledge base are available. The default degree serves as utility – the higher the default degree of a template the more likely it will be chosen in the end. Algorithm 11.2 shows only the part for operational mode. The complete algorithm implemented in the reasoning plugin for the adaptation middleware also includes the part for manual adaptation in training mode. We have seen this already in Algorithm 10.1 and omit for reasons of clarity at this point.

11.3.7 Example

To exemplify the learning-based adaptation reasoning, we will present a theoretic example and simulate the training and classification procedure. Table 11.2 shows a knowledge base after a short training period. The situations Working, Ontheway, Night, Infrastructure, and Activity are taken from the overview in Figure 11.6. We assume that these situations have been defined by appropriate rules. From the overview we took the four application variants High Performance, Low Performance, Navigation, and Information, which have been sampled two times by the user, except the High Performance variant which has three recorded samples.

| Table 11.2: Example of a knowledge base after short training. |
|---------------------------------|----------|----------|---------|---------|----------|
|                                  | Working  | Ontheway | Night   | Infrastr.| Activity |
| High Perf.                       | 1.1      | 1.8      | 1.0     | 2.0      | 2.1       | 3        |
| Low Perf.                        | 0.4      | 1.8      | 2.0     | 0.1      | 0.1       | 2        |
| Navigation                       | 0.1      | 1.8      | 0.0     | 0.2      | 0.8       | 2        |
| Information                      | 0.0      | 1.8      | 0.4     | 1.0      | 1.0       | 2        |

Assuming we switch to operational mode with the knowledge base from Table 11.2. Let us further assume that we are currently sitting in a pub and we would like to have the best configuration for the application in this situation based on the trained knowledge base. The first row in Table 11.3 specifies the current situation vector describing the pub situation, e.g. the pub situation is 0.1 of the Working situation but 0.99 of the Night situation.

| Table 11.3: Classification based on the observed data using the k-NN algorithm. |
|---------------------------------|----------|----------|---------|---------|----------|--------|
|                                  | Working  | Ontheway | Night   | Infrastr.| Activity |
| Pub Situation                    | 0.1      | 0.1      | 0.99    | 0.1      | 0.1      |
| △ High Perf.                    | 0.248    | 0.480    | 0.584   | 0.504    | 0.504    | 2.319  |
| △ Low Perf.                     | 0.064    | 0.288    | 0.000   | 0.010    | 0.010    | 0.371  |
| △ Navigation                    | 0.010    | 0.288    | 0.000   | 0.000    | 0.288    | 0.586  |
| △ Information                   | 0.000    | 0.288    | 0.506   | 0.400    | 0.400    | 1.594  |

The first step is to calculate the mean value for each of the entries by dividing an entry by the sample count. This mean value is used as input for the weighing function. Last but not least, the weighted mean value is subtracted from the current situation.
vector value. For example, the mean value for the High Performance variant in the Night situation is 0.6, which is close to 0.5. Thus, the weight is 0.96 resulting in a difference of $0.96 \times (0.6 - 0.1) = 0.48$. Finally, all differences are added up to a total sum. The application variant with the lowest sum (i.e. the nearest neighbour) will be taken. In this example, the Low Performance variant would be chosen.

**Algorithm 11.2** Adaptation reasoning based on the k-nearest neighbour classification.

```plaintext
function REASON
    for all applications do
        create TemplateSet
    end for
    create TemplateSetEnumerator
    while TemplateSetEnumerator.hasNext do
        currentApplicationSet ← TemplateSetEnumerator.next
        CheckResourceLimits(currentApplicationSet, availableResources)
        CheckArchitecturalConstraints(currentApplicationSet)
        if knowledgeBase == empty then
            continue
        end if
        weightedUtility ← 0.0
        currentSituation ← knowledgebase(currentSituation)
        for i ← 0, applications do
            templateID ← currentApplicationSet[i].identifier
            situationVector ← knowledgebase(templateID)
            if situationVector NOT null then
                for k ← 0, situationVector do
                    x ← situationVector[k]/counter
                    func ← $(4 \times x \times x - 4 \times x + 1) \times |x - currentSituation[k]|$
                    rawUtility ← rawUtility + func
                end for
            else if defaultMode then
                rawUtility ← GETDEFAULTDEGREE(currentApplicationSet[i]) - 1
            else
                rawUtility ← 0.0
            end if
            weightedUtility ← weightedUtility + rawUtility + priority
            if rawUtility == 0.0 then
                continue
            end if
        end for
        if weightedUtility ≤ bestUtility then
            bestUtility ← weightedUtility
            bestSet ← currentApplicationSet
        end if
    end while
    return bestSet
end function
```

146 Behavioural User Participation
11.3.8 Summary

With learning-based adaptation reasoning, we enable user participation in a more implicit way, but reduce the actual need for user participation significantly at the same time. If done well, learning the user’s behaviour is a good way to address uncertainty in the target group of the software, uncertainty due to concept drift, or uncertainty due to false assumptions.

By choosing a k-nearest neighbour classification algorithm, we employ a learning mechanism that is simple and efficient to implement, comprehensible from a developer and user perspective, and able to produce results with only little training effort. Further, the knowledge base can also be trained and adjusted during operational mode. The underlying MUSIC context ontology is a powerful concept to address sensor information. The two step approach is independent of concrete sensor implementations, i.e. sensors can be exchanged or re-defined in case of larger changes. The fuzzy inference from context sensors to situations allows simple and robust reasoning in the first step. The number of defined situations can be changed and existing situations can be adjusted at run-time to better reflect the user’s habits.

The presented approach has a few drawbacks. First, it requires the manual adaptation mechanism presented in Section 10.3 and the feedback manager described in Chapter 9. Second, to achieve proper learning of as many application variants as possible, the different variants must be selectable from the application’s user interface. This raises new challenges from an interface design perspective and is also costly and time consuming to implement. We could reduce this effort by limiting the number of application variants to those that contain user focus plans only (cf. Section 10.5), so that only template hierarchies with distinct user focuses can be used during training. And third, the performance of the classification depends on the defined situations. If the situations do not reflect the user’s environment and habits, the adaptation reasoning will not bring adequate results. We expect users to be able to specify situations properly. The implemented user interface design for the Android system shows the possibility of simple situation management. Although the process by defining IF-THEN rules is rather simple, it still requires the user to be actively involved. Alternatively, a developer or application maintainer could specify universal situations in advance. However, there still might be a gap between the developer’s considerations and the user’s setting.
Part III

Evaluation
12 Case Study: Meet-U

In this chapter we present the Meet-U case study. Meet-U [214, 215] is a self-adaptive and mobile application for social networking and event planning. We developed Meet-U to demonstrate the idea of smart mobile applications that employs an architectural adaptation approach. Meet-U was subject of several multidisciplinary studies before, e.g. Söllner et al. [139] studied Meet-U regarding trust formation. In this chapter we introduce the Meet-U application and two other studies on the effects of adaptations.

12.1 Meet-U

The first prototype of Meet-U requires the MUSIC middleware and was developed with the MUSIC methodology for self-adaptive applications [215]. With Meet-U, users can organise meetings with friends that take place at public or private events, such as movies at cinemas or birthday parties. Meet-U supports users in planning and managing events, in navigating to an event, and in being at that event. It has been designed to support its users in every situation by adapting to dynamic context changes. Further, Meet-U embeds the concept of adaptation to external services and is able to bind services provided by event organisers or other environmental services. Such services offer new valuable functionality such as information about movies or the possibility of buying tickets.

![Registration.](a) Dashboard. (b) Event details. (d) Navigation.

**Figure 12.1:** Details on the Meet-U application: users can create events, invite friends to events, navigate to events and use provided services.

Later on, a second prototype of Meet-U [214] was developed employing a multidisciplinary development process emerged from project VENUS [32]. The focus of the
**VENUS development method** was to integrated research from trust engineering, usability engineering, law, and computer science into a development method for ubiquitous computing applications. The second prototype of Meet-U had the same functionality as the first prototype, but was developed considering the multidisciplinary input. However, most changes affected the user interface only as many of the requirements from the other disciplines address what users actually see, i.e. design, usability, or workflow. Figure 12.1 shows four screenshots of the second Meet-U prototype. The first illustrates the law-compatible registration process. The second depicts the so-called dashboard, which is the main screen of the application. The third picture represents the dialogue when editing event information for self-created events. On the last screenshot we can see the navigation mode with information on the route and the event itself.

Using the elaborate MUSIC adaptation concept with its model-driven design philosophy, Meet-U tries to anticipate user actions and autonomously adapts to new situations according to the current context. Figure 12.2 illustrates examples of variability in Meet-U. On the first level (Figure 12.2 a)), four different realisation plans for the Meet-U application exist: *Offline*, *Planning*, *OnTheWay*, and *AtEvent*. Each plan represents one of the main application modes. Depending on component properties, context information, and the utility function, the best suitable plan hierarchy (i.e. application variant) is selected. Hence, Meet-U employs a functional adaptation concept as opposed to non-functional QoS optimisation. We will now describe each of the plans more detailed:

**Offline.** Meet-U requires several external web services (*event service*, *friend service*, and *transportplanner service*) for its normal operation modes. The *event service* connection exchanges event information with a central server. Similar, the *friend service*, which does the same for friend management. The *transportplanner service* calculates routes depending on the specified means of transportation. Meet-U relies on these services and hence a working Internet connection. If no Internet connection is available, *Offline* mode is selected by the adaptation middleware. Only the log-in screen and basic information are provided in this application mode.

**Planning.** The *Planning* realisation is the standard application mode. It allows users to manage the friend list, invite new friends, and react to invitations from new friends. Further, users can create new events and invite friends to these events. A search interface allows users to look for public events or events created by friends. For the latter, so-called invitations are created which the user can accept or decline. The planning mode does not have further adaptation capabilities.

**OnTheWay.** Depending on the user’s settings, Meet-U calculates the moment when it is time to start going over to the event’s location. The approximation is based on the starting time of the event, the distance to the event, and the means of transportation. By default, the heuristic projects the time of arrival five minutes before the start of the event. Then, Meet-U adapts to the navigation mode and calculates a route depending on the configured means of transportation. The *OnTheWay* mode provides more variability by adjusting its navigation to new situations, e.g. when the user arrives at locations for which more detailed routing and map information is available. This information can be either built-in as different components or it can be integrated from external services. For example, we developed an indoor navigation service that makes use of RFID-based localisation. Furthermore, Meet-U can select between different built-in map providers like Google or OpenStreetMap.
(a) Application type, first-level realisation plans (application modes), and static utility function.

(b) Composite realisation for the Planning mode that again has multiple other component types.

(c) Composite realisation for the AtEvent mode that again has multiple other component types.

Figure 12.2: Examples for different types of variability in Meet-U.
AtEvent. The AtEvent realisation should be selected by the adaptation middleware when the user arrives at the event’s location. If available, Meet-U integrates external third party AtEvent services. For example, cinemas can provide ticket or information services. If no such service is available, an internal component is selected that provides static information or if set, the website of the event. Depending on the type of event, the sound volume of the device is automatically adjusted. For example, if the event type is cinema, the device will be muted and if the type is party, it will be maximised. Furthermore, the AtEvent realisation includes a ContentSharing part which enables friends to share messages, pictures, or video clips. However, the content sharing feature was dropped for the second prototype.

As each of the four realisation plans on the first level is a composite realisation, it consists of several more component types. The composite realisation for the Planning mode is shown in Figure 12.2 b). It includes the four component types UI, Controller, Calendar, and UserProfile. Each of these component types must again be realised by at least one realisation plan, i.e. the concept of compositional adaptation is applied. Service connections delegated from the composite Planning realisation to the Controller component type employ the service adaptation concept. Each of the service port types have to be realised by any number of service instances. For example, the eventservice_req port binds the external event service. A composite realisation can only be realised if all delegated port dependencies are met. Figure 12.2 c) depicts the composite realisation for the AtEvent mode. The differences compared to the Planning realisation are in the including component types and service connections. The AtEvent mode does not require the calendar component type which was used in Planning mode, but instead the ContentSharing component type is included to enable content sharing among users. The DeviceController type manages the parametric adaptation that controls the sound volume of the device at the event location. A transportplanner service is not required, but instead a third party AtEvent service can be bound. However, if such service is not available, a local (optional) instance of the AtEventComponent will be used so that the delegated port dependency is met.

All component types and their realisations as well as the compositions can be seen in Section D.1 of the appendices. The total number of application variants results from the number of possible plan combinations. That means, there are at least four application variants resulting from the first-level realisation plans. This number increases by different available component type realisations within the four compositions. Furthermore, parameter settings as used in the DeviceController of the AtEvent realisation, multiply the total number of variants quickly (cf. Figure D.3 in Appendix D).

12.2 Initial Study

In a qualitative user analysis with eight potential users\(^1\), the usability of Meet-U and its adaptations was evaluated before we applied the VENUS development method. The goal was to identify the effects of adaptations on the usability of an application. We already

\(^1\)Discount usability engineering [31] states that five participants are enough to detect as many usability problems as it could be found when using many more test participants [107]. A bigger number of participants lowers the benefit-cost ratio in qualitative user testing.

154 Case Study: Meet-U
noticed several usability limitations of the self-adaptation concept during development. Therefore, we needed to know whether the usability limitations would be experienced by ordinary users, too. The second objective was to uncover other issues regarding self-adaptation and usability in the practical use of Meet-U.

For the study we selected participants which indicated to have an interest in new mobile applications and social networking. All of them were mechanical engineering students at the University of Kassel. Participants of the study were male ($N = 7$) and female ($N = 1$) students with an average age of 23.4 years and had at least minimal experience with smartphone devices. The scenario-based field study took place in a laboratory as well as in a field setting on the university campus. The participants were given a scenario that put them into the situation of a new student who has been invited to a mentor's meeting at a laboratory of the university. The scenario contained different tasks. At first, a participant had to create an event and invite friends from the list of friends to the event. Second, the participant should walk over to the event location using Meet-U's built-in navigation mode. As the laboratory is located in another building, this task involved using the outdoor and indoor navigation capabilities of Meet-U. Once arrived at the laboratory, a participant had to find out particular information regarding a mobile robot located in the laboratory. The required information was provided via an external service which Meet-U integrated as soon as the participant entered the room.

An observer accompanied the participants during the study. They had to follow the method of Thinking Aloud in order that their experience with Meet-U was revealed to the observer and usability problems could be uncovered quickly and in-situ. However, the observer was not allowed to interact with the participant unless an error within the software occurred. At the end, a short interview was conducted. Each pass including the practical tasks and the interview took about one hour per participant.

The tasks assigned to the participant are closely coupled with Meet-U’s adaptations. Here, Meet-U provides three different adaptations which all belong to the class of functional adaptation:

1. Adapt to navigation mode (compositional adaptation): when the user necessarily had to leave for the event to be there in time, an adaptation was initiated that switches to the navigation mode. A map with additional routing information was displayed to support the user in finding the exact location of the building in which the laboratory is situated.

2. Adapt to indoor map (compositional or service adaptation): when the user continued his way indoors, the outdoor map was substituted by an indoor map of the building where the laboratory is placed. An optional picture of the event location should help finding the correct room. For the indoor map, an optional service can be bound that provides the necessary information.

3. Adapt to an external event service (service adaptation): as soon as the user had reached the laboratory, the _AtEvent_ service was integrated. Then, Meet-U provided detailed information about the laboratory and available mobile robots.

In the following, we summarise the results reported by the participants. Besides general usability issues, the evaluation revealed problems regarding Meet-U’s adaptations in particular. The majority of participants wanted to be reminded on an upcoming event in an adequate and user-defined period beforehand. This reminder would work as a forecast...
generating user expectations for the potential navigational adaptation and might reduce confusion about the suddenly showed map while doing something else (first adaptation). Moreover, the participants criticised that the first adaptation was not controllable at all. They wanted to be able to cancel the adaptation or exit the navigational screen. Both were not possible as the underlying component structure was determined by the MUSIC middleware. Participants also recommended that the information dialogue of the AtEvent service should always be manually accessible. During the study, this was only shown when Meet-U detected the service and the location and time matched (third adaptation). Further, participants complained about the spontaneity of the sudden and non-notified view change when the indoor map and the external service were displayed (second adaptation). All adaptations have not been notified beforehand. When the adaptation middleware decided to adapt the current application variant, it switches components without prior notification and without considering whether the user is currently interacting with the device or not.

Although this initial study was conducted with eight participants only, it showed significant problems with functional self-adaptation whenever users are involved. The next section presents the results of a more differentiated evaluation with a large sample size and the second prototype of Meet-U.

12.3 Evaluation Study

To get a more differentiated picture on the effects of self-adaptation, we conducted a large user evaluation with $N = 62$ participants [219, 222, 223]. This time, the second Meet-U prototype including the notification concept (cf. Chapter 9) with pre- and post-controllability (cf. Chapter 10) has been used. We were particularly curious how these newly developed concepts are approved by potential users. To be precisely, we formulated the following research questions which have been deducted implicitly or asked explicitly during the study:

Q1: Do notifications on upcoming or prior adaptations improve user acceptance?

Q2: If yes, what type of notification is preferred by the participants?

Q3: Do participants embrace or refuse autonomous adaptations in general?

Q4: What type of personal information would participants provide to make adaptations more user-oriented?

Q5: Do participants want to change the adaptive behaviour at run-time?

Q6: If yes, what type of interface do participants prefer?

We assume that users will not prefer a pure autonomous behaviour of applications and that there is not the one configuration that fits all users. Due to the individual differences among users, they will prefer to change any predefined behaviour. In the following, we will describe the evaluation method, the concrete realisation, and the execution of the study. We conclude this section by presenting the results to the above research questions.
12.3.1 Method and Realisation

Before we started with the actual study, qualified participants had to be acquired. Potential users had to fill in a screening questionnaire which took about 10 min per participant. Based on the results we chose $N = 62$ participants that fit our target group. During the requirements analysis phase of the VENUS development method, we identified the target group for the Meet-U application as smartphone users between 18 and 42 years old who like to keep in touch with regular friends by planning and joining common events. The study itself involved two consecutive parts: first, the participants had to accomplish several practical tasks while being supported by Meet-U. The tasks fitted in a surrounding scenario occurring on the university campus, which was explained to participants by a supervisor in preparation of the study. And second, participants were asked to answer questions based on the experiences they just had made with Meet-U. The second part was held in a laboratory. A complete iteration over both parts with one participant took about 1.5 hours. Participants were female ($N_f = 28$) and male ($N_m = 34$) with an average age of 24.3 years. Most of them were students ($N_{st} = 59$) of different fields of study, such as psychology, mechanical engineering, or computer science.

First part (hands-on experience). The scenario given to the participants put them into the role of a student who has recently begun to utilise Meet-U to plan, navigate to, and to take part in meetings with fellow students. The participants were asked to timely navigate to a lecture event, to integrate the event service of the lecture, and to reject the adaptation that mutes the mobile device in the lecture room. The lecture took place in a building and an area most participants were unfamiliar with. Participants started their tasks in the university canteen for which we simulated a menu service. For the participants it seemed that the service was provided by the canteen. Within the building in which the lecture took place, an indoor navigation service helped participants in finding the event room. The five included adaptations (A1-A5) were partially already included in the first initial user study. These adaptations are:

A1: Service adaptation to the canteen menu service. Meet-U displays the daily menu of the canteen. The service is provided by the canteen. This adaptation is based on the AtEvent service composition. We call this type of service environmental as it does not provide information relevant to the current task of the user, but is still provided within the vicinity of the user.

A2: Compositional adaptation to the outdoor navigation mode. Depending on the beginning of the event, the means of transportation, and the current time, Meet-U calculates the moment when it is best to leave for the event and helps navigating to the destination by providing a map and routing information. This adaptation is based on the OnTheWay composition. The navigation mode is used to guide the participant from the building where the canteen is located to the building where the lecture took place. The distance is about 400 m, including two turns.

A3: Service adaptation to an indoor navigation service. This adaptation is also based on the OnTheWay composition, but makes use of a local service providing indoor navigation information for a specific building. For the evaluation, a plan layout of the building was provided where the lecture took place, including room numbers and a clear declaration of the destination room.
A4: Parametric adaptation for muting the device at the lecture room. This adaptation is provided by the `DeviceController` nested in the `AtEvent` composition. It can control the sound volume of the device depending on context parameters like the current location.

A5: Service adaptation to an event service providing a schedule of the lecture course at the destination location. When the participants arrive at the lecture room, Meet-U integrates a service that provides additional information on the lecture. This information includes a detailed time schedule and descriptions on the topic. Although adaptations A1 and A5 are different adaptations within the evaluation scenario, they both use the `AtEvent` realisation of Meet-U.

We created five different design variations (V1-V5) based on the notification concept implementing pre- and post-controllability from the temporal participation dimension (cf. Chapter 9). Figure 12.3 shows the concrete design variations for the outdoor navigation adaptation (A2). As we have seen in Section 9.1, the variations differ in degree of controllability (none, post, or pre) and type of notification (none, annunciator row, notification strip, or pop-up). As described in Section 10.2, post-controllability lets the user decide what to do after an adaptation was initiated. Pre-controllability asks the user in advance, i.e. before adapting the application (cf. Section 10.1). The easier an adaptation can be manipulated or controlled, the more controllable it becomes (cf. Section 9.1). Therefore, pre-controllability has a higher controllability than post-controllability. Each design variation uses a vibration pattern and a sound which clearly indicate the occurrence of an adaptation. The supervisor introduced both to the participant while describing the scenario at the starting point (in the canteen).

Design variation V1 is the reference design for autonomous adaptations as it does not implement any user participation concept. Each design variation was evaluated by a single group (G1-G4) of participants (N = 13). However, V2 was only presented during the laboratory part of the study and had no group associated in the hands-on session. A group G5 was confronted with different design variations for each of the adaptations. The selection of a design variation for a particular adaptation was based on the results from groups 1 to 4. For each adaptation the previously best-voted variant was chosen. The participants were randomly assigned to the groups. Only the number of female and male participants was controlled in order to reach an equal distribution over all groups. Table 12.1 summarises the association of design variations, groups, and adaptations.

### Table 12.1: Assignment of groups of participant to design variations and adaptations. Each group was assigned to one design variation. G1 is the reference group with no controllability and G5 had different design variations for each of the adaptations.

<table>
<thead>
<tr>
<th>Controllability</th>
<th>Design Variation</th>
<th>Group (N_i = 13)</th>
<th>Adaptations Assigned in Group G5 (N_5 = 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>No Notification (V1)</td>
<td>G1</td>
<td>–</td>
</tr>
<tr>
<td>Post</td>
<td>Annunciator Row (V2)</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Notification Strip (V3)</td>
<td>G2</td>
<td>Indoor Navigation (A3), Device Muting (A4)</td>
</tr>
<tr>
<td>Pre</td>
<td>Annunciator Row (V4)</td>
<td>G3</td>
<td>Environmental Service (A1)</td>
</tr>
<tr>
<td></td>
<td>Pop-Up (V5)</td>
<td>G4</td>
<td>Outdoor Navigation (A2), Event Service (A5)</td>
</tr>
</tbody>
</table>

158 Case Study: Meet-U
Although fully functional and working, Meet-U was evaluated as a Wizard of Oz experiment\(^2\). In the first initial study we noticed few technical issues with lost network connections and failed handovers between the wireless and the cellular network. And as we have motivated extensively, there were also problems with run-time uncertainty

\(^2\)In a Wizard of Oz experiment the participant thinks the system is acting autonomously although it is (partially) controlled by a human being – the wizard.
of the self-adaptation concept. These problems were not in the focus of this evaluation and participants should not become distracted. Hence, we made use of the *Wizard of Oz* experiment setting to eliminate the risk of wrong adaptations and to react on sudden changes. The wizard was played by the supervisor who triggered adaptations remotely. The participants were not informed about the fact that the adaptations were initiated by the wizard in order to avoid any influence.

The *Wizard of Oz* concept was implemented using a remote control device. The device was operated by the supervisor who also was responsible for the regular execution of the study. Meet-U was extended to be externally controlled by another device. With the remote control all possible adaptations with all possible design variations can be executed. For this study, only a subset of adaptations was selected to keep the handling clear for the supervisor (cf. Figure 12.4 b)). The different notification variations have to be configured for each adaptation within the Meet-U application in advance (cf. Figure 12.4 c)). Vibration pattern and sound can be set optionally for each adaptation. The remote control can be used for multiple running instances of Meet-U (cf. Figure 12.4 a)).

![Figure 12.4: The remote control application and the notification settings within Meet-U.](image)

**Second Part (laboratory setting).** After the participants finished the tasks, they were asked to assess the general usability of the application, the adaptations, and the services they just had experienced. In the beginning of this part, the supervisor of the evaluation completed a prepared observation protocol. During this time, participants completed a questionnaire on the general acceptance of Meet-U and its services, Meet-U’s general usability, and in particular the effect of adaptations on the application’s usability. Questions regarding the latter are relevant for analysing user participation in adaptive applications. Further, participants were asked to vote for a preferred design variation for each adaptation and to motivate their decision. The specific usability analysis and other aspects concerning the interaction design of Meet-U are not part of this work and we refer to Kniewel et al. [222, 223].

160  Case Study: Meet-U
In this section we analyse the results of the design variation voting and the answers to the questions asked during the second part of the study. A central question is whether notifications on upcoming or prior adaptations improve user acceptance or not. During the second part of the study, we let participants vote which design variation they thought is best, respectively liked most, for a particular adaptation. Hence, our assumption is, that user acceptance directly correlates with the voting of a participant. Further, we assume that the participant's choice depends on the degree of controllability. Controllability is defined by the following indicators: first, a design variation has the possibility to control the adaptation, second, the quantity of dialogue interactions required to control the adaptation (direct versus indirect controllability), and third, the moment of control (pre-controllability versus post-controllability). Figure 12.5 a) depicts which of the design variations have been chosen by participants for the different adaptations. The diagram points out that the chosen design variation depends very much on the type of adaptation. Thus, we cannot generally claim for the same design variation for all adaptations and we need a more detailed look at the adaptations and the design variations used.

Therefore, Figure 12.5 b) categories the variations into such with direct control (V3 and V5) and such with indirect control (V2 and V4). Direct control requires one interaction step to complete the adaptation interaction, indirect control requires more than one step and is hence less controllable. For example, the design variation Annunciator Row has a lower controllability than the variation Pop-Up because more interaction steps are needed to finish the adaptation interaction. The autonomous variation V1 provides no control at all. Except for the canteen and event services, participants prefer a more direct control on the adaptations. This can be explained by the fact that the participants do not want to be interrupted by an adaptation not supporting the currently conducted task. In the canteen, the participants were asked to informally chat with the supervisor. Thus, there was no need to use the canteen service. In contrast, the two navigational adaptations supported the participants’ current tasks (participants had to be at the event on time). Hence, they would like to be interrupted from a current activity and be directed to a more immediate task (navigating to the lecture). Additionally, the indoor navigation adaptation was executed when a participant was already using the application for navigation to the unknown lecture room in the unknown building, which is a similar task. No control was only chosen by a significant number of participants for the indoor service adaptation. An often mentioned reason was that they were already been navigated anyway and they did not know the way within the building and asked for support. It was noticed that participants liked the fact that Meet-U anticipated the indoor navigation situation and made a reasonable decision.

An analysis of pre- and post-adaptation control is depicted in Figure 12.5 c). Again, it can be seen that participants voted for more controllability. In particular pre-controllability would be chosen, if the adaptation did not support the current task. It can be also inferred that adaptations with a strong influence on the user interface and the interaction flow (A1, A2, and A5) should be realised with pre-controllability. Others should at least use post-controllability. No controllability is only desired by very few participants. Generally speaking, adaptations supporting the user’s current task instead of distracting him are more likely to be accepted. Hence, we imply successful adaptations as controllable and task-supportive.
**Figure 12.5:** Which design variations did participants prefer for which adaptation?

**Case Study: Meet-U**
In contrast, the answers to research question Q3 reveal a very mixed picture (Figure 12.6). 25 participants would rather not or definitely not use an application that performs adaptations autonomously while 31 participants would rather use or definitely use such an application. Another six participants are not sure whether they would. Despite the fact that participants prefer design variations with more control, they do not entirely disagree on autonomously applications. We can see that V3 and VM (the mixed and optimised set of design variations for group 5) had about 60% of participants each who decided in favour of autonomous adaptations. These are either design variations with a high degree of controllability (V3) or were the optimised set (VM) of design variations. This result is not very reasonable. We assume that participants misunderstood the question and implied autonomous adaptations to be adaptations with notifications.

![Figure 12.6: Would participants use an application that works only autonomously? VM represents group 5 which used a mixed and optimised set of design variations.](image)

**Figure 12.7:** Which personal information would participants provide to improve self-adaptive behaviour?

With user interaction activity detection (cf. Section 10.6) we developed a mechanism to optimise adaptation decisions based on the user’s current interaction activity with the
device. However, to classify the current activity we have to interpret personal context information. Therefore, we asked the participants what type of information they would reveal to optimise the adaptive behaviour of an application. The results are depicted in Figure 12.7. It is good to see that the large majority would reveal personal information as long as it is used for the adaptation optimisation only. About one half of the participants would reveal helpful data on the current smartphone usage. Nowadays, people grow up with applications that use their current location to provide services. This is clearly reflected by our study where about 75% would allow using the current location for interaction activity detection.

We assumed that people are not content with the predefined configuration and application behaviour. Hence, research questions Q5 and Q6 should reveal if the participants want to make changes in the behaviour and the dialogue properties. Figure 12.8 draws a clear picture: people definitely would like to have the possibility of individualising adaptations, e.g. with a construction kit for preferences and rules. In total, 59 participants either said yes or rather yes when asked for individualisation. The small rest was unsure whether they need it or not. When looking at each of the groups we see a more differentiated picture. Design variations 1, 4, and 5 created the most need for individualisation. All participants of these groups voted for at least rather yes. However, the relative fraction
of yes voters is high within the mixed design variation (VM), too, although we optimised this variation in regard to the adaptations. The two extreme variations (V1 and V5) got the most yes votes absolutely. The least need for individualisation was for variation V3. As a result we have to say that there is no clear tendency in regard to the used design variations. When it comes to the type of interface we let participants chose between building blocks and behaviour profiles. While building blocks allow a more detailed configuration of the application behaviour, profiles embrace a few predefined application configurations. From Figure 12.9 we can deduct a preference for building blocks (42 participants) compared to profiles (13 participants). Seven participants could not decide which type to choose.

When answering this question, participants often commented that they surely want to dictate how the application behaves or at least have the chance to make changes. This is somewhat contradictory when compared to many successful mobile (non-adaptive) applications which are usually simple and do not provide many adjustments to allow users easily grasp the idea of the application. However, the results make clear that we need explicit user participation in adaptive applications by letting users define preferences and adaptation settings.

12.4 Summary

The Meet-U case study is a well-suited example to demonstrate functional adaptation in mobile applications. It consists of several application modes implementing the compositional adaptation scheme representing strong adaptations. The integration of external services allows for an easy extension of functionality. During the development of the first prototype we have noticed several usability and interaction design issues caused by the underlying MUSIC adaptation middleware and the corresponding methodology. A qualitative user analysis using discount usability engineering confirmed our findings.

We developed a second prototype of the Meet-U application employs the VENUS development method, which has a special focus on requirements from trust engineering and law. However, the issues caused by adaptations were not addressed properly. This was the motivation to apply the notification and feedback as well as pre- and post-controllability concepts to the Meet-U prototype. This extended version of the second Meet-U prototype was subject of a second larger user study with 62 participants. The total duration of the study covered about nine month including preparation and analysis. Five different design variations were applied to five different adaptations which supported participants in accomplishing their tasks. In the end of a practical session, participants had to vote for design variations which they liked most. As a result we can say that there is not the one design variation that fits all types of adaptations. Instead we must look at the individual situation a user is in and which task he is currently pursuing. If the adaptation supports the task, a user would prefer a more autonomous adaptation with a notification afterwards stating the reason for adaptation. If the adaptation is orthogonal to the current task (e.g. environmental service like the canteen service), the user would prefer a more unobtrusive notification beforehand. These results bring up new challenges for the requirements engineering process; analysts and designers have to look very detailed at every use case. However, we have clearly seen that the design variation with full autonomous adaptation, was only preferred by a minority of
participants. The majority of participants voted for design variations that make use of the notification concept, independent of the amount of information, controllability, or obtrusiveness.

Further, the participants made clear that they want to individualise the application behaviour in regard to their needs. We assume that participants noticed a slightly improper behaviour of the application during the hands-on experience with Meet-U. Although participants were not clear about the type of individualisation, we have presented three possible ways of behavioural modification in Chapter 11. These have to be evaluated in another study.

Although very interesting for real context-aware and adaptive applications, we did not evaluate whether the moment of adaptation was appropriate or not. This was because of the study design and the decision to let the supervisor initiate all adaptations with the external remote control. However, user interaction activity detection as presented in Chapter 10 could support the adaptation middleware to choose the right moment for adaptation. From the questionnaire we know that people would provide the required personal information.

The results of the study confirm our assumption and motivation for more user participation in self-adaptive software. There are still some other possible improvements for the Meet-U application regarding the adaptation behaviour. The consideration of the user focus (cf. Section 10.5) would help to decide which design variation to take for a particular adaptation.
13 Demonstration Applications

Besides the extensive Meet-U case study presented in the previous chapter, we developed six demonstration applications to evaluate the practical feasibility of the user participation mechanisms. The applications were developed as OSGi bundles to be installed in the MUSIC middleware with user participation support. As the MUSIC middleware also runs on the Android operating system within an OSGi run-time environment, applications can theoretically run on the Android system, too. However, Android does not have native MUSIC support. Every MUSIC application requires a corresponding Android application on top which provides the necessary user interfaces. For most of the mechanisms for user participation there is no need to be demonstrated on the Android operating system. Demonstrator C (interaction activity) runs on Android only, as it makes user of the newly developed interaction activity detection sensors. It is the only demonstrator with an additional Android application on top. All other applications were tested on the Windows or Linux versions of the extended MUSIC middleware. However, they would run within the Android version of the middleware, too, but have no graphical user interface attached. Table 13.1 summarises the demonstration applications and shows which user participation mechanism is addresses by which demonstrator. Demonstrators A, B, and C address the temporal participation dimension whereas demonstrators D, E, and F address the behavioural participation dimension.

Table 13.1: The different demonstrators and the mechanisms for user participation they realise.

<table>
<thead>
<tr>
<th>Demonstrator</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-/Post-Controllability</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manual Adaptation</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pausing/Stopping Adaptation</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User Focus Adaptation</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interaction Activity Detection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic Utility Functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fuzzy-based Reasoning</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Learning-based Reasoning</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All demonstration applications are based on the HelloWorldApp first introduced in Chapter 3. The original component types and realisations are depicted in Section D.2 of the appendices. This simple example is complex enough to implement compositional adaptation and is yet capable of demonstrating the mechanisms for user participation. The overall goal of the HelloWorldApp application is to display the text Hello World on the screen. The text should be always readable horizontally, no matter how the user holds the device. Thus, the adaptation middleware adjusts the application according to the screen...
orientation. Moreover, the amount of required memory for each of the components is used as self-information. The application consists of two possible realisations, a SimpleHelloWorld atomic realisation and a ComplexHelloWorld composite realisation. SimpleHelloWorld uses parametric adaptation and provides already a valid solution to the above problem description. Further, it has a very low memory requirement of 20 kB for mobile devices with little memory. If devices have a larger amount of memory, the ComplexHelloWorld composite realisation can be used. It separates the application logic from the user interface by including two component types for each. There are two realisations for the UI_Type, one for portrait and one for landscape mode. The AL_Type has also two realisations with different memory demands. The utility function evaluates all application variants and determines the utility based on the current orientation of the device (context). The memory consumption (self) is evaluated during the resource check of the adaptation reasoner and is not considered by the utility function itself. Although this example is rather complex for a hello world example, it allows the demonstration of the different mechanisms for user participation.

To allow reproducibility of test cases, we provide the middleware configuration in terms of required OSGi bundles in Table 13.2 at the end of this chapter. The build automation tool Apache Maven is used to build the OSGi bundles for the different operating systems.

13.1 Demonstrator A: Interaction

The interaction demonstrator uses the basic HelloWorldApp example without major changes to the underlying variability model and application structure. The demonstrator shows pre- and post-controllability, manual adaptation, and pausing adaptation. All mechanisms belong to the temporal participation dimensions. This demonstrator requires either adaptation.plugins.manual or adaptation.plugins.userfocus as adaptation algorithm (cf. Table 13.2).

<table>
<thead>
<tr>
<th>SimpleHelloWorld</th>
<th>PortraitUI</th>
<th>LandscapeUI</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendFeedback=true</td>
<td>receiveNotifications=true</td>
<td>property={landscapeProvided=true, cPSService=15000} sendFeedback=true receiveNotifications=true</td>
</tr>
</tbody>
</table>

Figure 13.1: Demonstrator A: User interface realisations with notification and feedback.

Furthermore, the above mechanisms require bi-directional communication between the middleware and the application, i.e. notification and feedback components. Therefore, we have to mark the component realisation within the variability model accordingly. Figure 13.1 shows the application realisation SimpleHelloWorld and the two user interface realisations LandscapeUI and PortraitUI. All three have been defined as notification receivers and feedback components within the model. Although SimpleHelloWorld is not a user interface realisation in the explicit definition, it represents a single application variant with two parameter settings on its own and consequently an implicit realisation of a user interface. However, for the basic example we renounced the use of a real user interface implementation. The different mechanisms are tested within the component stubs using the same hard-coded method calls as they would be used by a graphical user interface.
Listing 13.1: Demonstrator A: The PortraitUI implementation using the notification and feedback mechanisms as well as simulated method calls for sending feedback and handling notifications.

```java
public class PortraitUI extends ConfigurableImpl implements IUI, IConnectable, INotificationClient, IFeedbackComponent {

private IFeedbackManager fm;
private MusicName myname;

public PortraitUI() {
    super();
    System.out.println("PortraitUI created");
}

public void startActivity() {
    super.startActivity();
    MusicName UI_Type = MusicName
        .nameFromString("/type/de.unikassel.vs.demonstrator.e/UI_Type");
    fm.preferComponent(UI_Type, new String[] { "LandscapeUI" });
    //fm.handleFeedback(myname, FeedbackType.PAUSE_ADAPTATION, new Long(5000));
}

public void displayData(String input, int type) {
    System.out.println("PortraitUI control:");
    System.out.println(input.replace(' ', '
'));
    if (type == 1)
        fm.handleFeedback(myname, FeedbackType.UNDO_ADAPTATION, null);
}

public void handleNotification(int type, String message) {
    System.out.print("PortraitUI: Notification from middleware received:");
    switch (type) {
        case NotificationType.HIGHER.Utility.PREFER:
            fm.handleFeedback(myname, FeedbackType.UNPREFER_PLAN, null);
            break;
        case NotificationType.HIGHER.Utility.UNDO:
            fm.handleFeedback(myname, FeedbackType.IGNORE_UNDO, null);
            break;
        case NotificationType.UTILITY_UNCLEAR:
        case NotificationType.UTILITY_DIFF_LOW:
            System.out.println("There is no best application variant");
            break;
        default:
            break;
    }
}

public void setFeedbackManager(IFeedbackManager fm, String myname) {
    this.myname = MusicName.nameFromString(myname);
    this.fm = fm;
    if (this.fm != null) {
        System.out.println("FeedbackManager successfully set in "+ this.getClass().getSimpleName());
    } else {
        System.out.println("Error in setFeedbackManager...null");
    }
}
```
Listing 13.1 shows the PortraitUI class as an example for a user interface component. The two interfaces INotificationClient and IFeedbackComponent are implemented in line 2. They allow the component to communicate with the middleware. While the component is automatically registered as notification receiver, the feedback manager has to be set explicitly during the configuration steps of the middleware (line 52). After component initialisation, the startActivity() method is called. In PortraitUI, we simulate either manual adaptation by preferring the LandscapeUI component (line 16) or we simulate pausing of adaptation by 5 seconds (line 17). Line 28-50 show examples for pre- or post-controllability (the exact mode depends on the configuration of the middleware). Two types of notifications can be handled here: first, the user did a manual adaptation before (plan preference), but another application variant has a higher utility then the current application variant (line 34). And second, the user has previously undone an adaptation, but now there is an application variant with a higher utility available (line 38). In case the utility is unclear or the difference between the best and the second best application variant are too low, a message is printed (line 42). The displayData(...) method in line 20 is called by one of the AL_Type realisations whenever the text Hello World should be displayed. This message carries an additional type and whenever this type is 1, the middleware is notified to undo the last adaptation. For example, whenever the AppLogic2 component is initialised it may send type = 1 along with the message (not shown here).

Although the different events are all simulated by hard-coded method calls, we demonstrate the proper working of the adaptation middleware with user participation capabilities.

### 13.2 Demonstrator B: User Focus

![Diagram of user interface realisations marked as user focuses.](image)

Figure 13.2: Demonstrator B: User interface realisations marked as user focuses.

Unlike Demonstrator A, this demonstrator requires adaptation.plugins.userfocus as the active adaptation algorithm (cf. Table 13.2). Again, little changes must be made to the variability model. We have to specify which of the components constitute a user focus. A user focus component cannot be adapted by the middleware autonomously. Components should belong to a single user focus if they are required to fulfil an independent function from a user perspective. For the HelloWorldApp example the independent functionality is in the different user interface realisations. Hence, each of the user interface components is a single user focus (cf. Figure 13.2). As SimpleHelloWorld also contains two parameter
settings for the user interface, we should mark this as a user focus, too. But instead of annotating the atomic realisation, we have to mark each of the realisation variants describing the parameters settings as depicted in Figure 13.2.

In combination with the adaptation.plugins.userfocus adaptation algorithm this is sufficient to enable user focus reasoning. However, to override user focus reasoning, i.e. to prefer other application variants and return back to user focus reasoning again, we need the notification and feedback concept as presented with Demonstrator A. Like the first demonstrator, we use simple component stubs with simulated method calls to demonstrate the mechanisms provided by the middleware. Listing 13.2 shows the relevant notification handling with the corresponding feedback to be sent. The user can be notified that another application variant has a higher utility than the currently active user focus variant (line 7). By letting the middleware know to ignore the user focus, the variant with the higher utility is chosen (line 8).

13.3 Demonstrator C: Interaction Activity Detection

The goal of the interaction activity demonstrator is to show how we can use the interaction activity level of a user as context information. As we have proposed in Section 10.6, this information can be used to adjust notification behaviour of the adaptation middleware. However, at first we have to make sure the presented activity classification algorithm is working.

Unlike the first two demonstrators, Demonstrator C has been developed for the Android operating system only. That means, not only the required MUSIC OSGi bundle was implemented, but also an Android application has been built on top to make changes in user interaction activity visible. Further, for this demonstrator the bundle context.plugins.sensors.android is required. It contains all available MUSIC context sensors for the Android system, e.g. battery, cell, GPS, NFC, Wi-Fi, or screen sensors. The context plugin for the user interaction activity is included, too. However, the actual activity sensor had to be put in an external library android.external because an Android application (the Android application hosting the MUSIC middleware within an OSGi environment)
requires specific system services in the same namespace, which the OSGi bundle for the android sensor plugins is not. The demonstrator is independent of any adaptation reasoning algorithms as we have not yet used the interaction activity classification to influence adaptation behaviour.

Figure 13.3: Demonstrator C: Using the interaction activity level for adaptation. The model shows the differences to the HelloWorldApp used by the other demonstrators. Instead of choosing a UI component as a function of the screen orientation, the UI component depends on the interaction activity level. The ComplexHelloWorld composition and the AL_Type realisations are the same as in the example applications depicted in Section D.2.

For this demonstrator, we had to change the HelloWorldApp example in a way that it can demonstrate the three levels of activity appropriately. The two UI_Types LandscapeUI and PortraitUI have been replaced by three realisations HighActivity, Activity, and NoActivity. Each of them has a property Activity with a value corresponding to the class of activity. Figure 13.3 illustrates the changed variability model of the application. We only use the ComplexHelloWorld composition and neglect the SimpleHelloWorld atomic realisation here. Moreover, the BatteryLoad context dependency and the PowerConsumption property are removed from the application. Landscape and LandscapeProvided are replaced by the dependency and property for the activity interaction level. The ActivityLevelQuery
context query defines the identifiers to access the interaction activity in the context ontology in terms of Entity, Scope, and ValueScope. The ActivityEvaluator depicted in Figure 13.3 propagates the property value for Activity from the UI_Type component to the HelloWorldApp application type, so it can be evaluated by the utility function. Without the property evaluator, the utility function could not access the Activity property of the user interface components. This is part of the hierarchical adaptation concept used by MUSIC and can be ignored for further understanding.

The corresponding Android user interface shows plain coloured screens in red for high activity, in green for undetermined activity, and blue for no activity. However, the activity class no activity includes the state when the display is switched off and no touchscreen inputs are registered. Thus, we cannot visualise if the interaction activity reasoner classified this state as no activity. Therefore, an external monitoring application has been developed to visualise the state of the interaction activity reasoner. It connects directly via a socket connection to the context plugin within the MUSIC middleware and does not depend on the Android application itself. The external monitoring software runs on Java Swing compatible devices. Figure 13.4 illustrates the Android demonstrator implementation and the external monitoring software.

13.4 Demonstrator D: Utility Building and Switching

The demonstration application for utility building and switching is based on the dynamic utility function concept. To allow for the dynamic run-time building of utility functions we have to provide the required meta data within the variability model. Dynamic utility functions are constituted of multiple sub-utility functions, each relating to a quality metric. Typically, a quality metric corresponds to a plan property in the original model.
That means, the two plan properties LandscapeProvided and PowerConsumption are extended to quality metrics. Figure 13.5 illustrates how quality metrics are modelled and which meta data we have to provide for the HelloWorldApp example. Each of the quality metrics is assigned to the corresponding context reference.

The LandscapeProvidedMetric relates to the Landscape context information. It should provide a high sub-utility value when the constant value of the metric and the context value match. Thus, we set evalType to Proportional. The data type for the metric and the context information is Boolean why have to use the tabular sub-utility function. The result of the proportional evaluation is either true (matches) or false (does not match). For this result we have to determine a utility value. In case they match, the utility will be high; otherwise it will be low.

The PowerConsumptionMetric relates to the BatteryLoad context information. It should provide a high sub-utility value when the battery load is high and the power consumption is high, too, i.e. they match and the evalType must be Proportional. Both values are within the real numbers and the proportional evaluator calculates the input value for the linear sub-utility function by establishing the absolute difference of the two values. The result is subtracted from the domain maximum value (here: 100) and fed into the linear utility function. A high difference between the battery load and the power consumption will result in a low utility value and a low difference will result in a high utility value. Finally, the dynamic utility function can be built and saved from the middleware’s user interface as pictured in Section 11.1.2. Users can then choose between the predefined static utility functions and previously created dynamic utility functions.

Figure 13.5: Demonstrator D: Specification of the two quality metrics LandscapeProvidedMetric and PowerConsumption replacing the plan properties LandscapeProvided and PowerConsumption.

To demonstrate the switching of static utility functions, we also added a second static utility function, which inverts the behaviour of the first one. That means, instead of adjusting the text output to match the screen orientation, it displays the text the opposite way. For example, when a user holds the device in portrait mode, the text is printed horizontally. Although, this is not reasonable for practical usage, it allows us to present the concept of utility switching.
Listing 13.3: Demonstrator D: Two static utility functions with UtilityFunction1 being the default function.

class UtilityFunction extends AbstractPropertyEvaluator implements IStaticUtilityFunctionContainer {
    private HashMap<String, AbstractPropertyEvaluator> functions;

    public UtilityFunction() {
        functions = new HashMap<String, AbstractPropertyEvaluator>();
        functions.put("UtilityFunction1", new AbstractPropertyEvaluator("UtilityFunction1") {
            public Object evaluate(IContextValueAccess context, IPropertyEvaluatorContext evalContext) {
                // evaluation code
                double utility = ...
                return new Double(utility);}});

        functions.put("UtilityFunction2", new AbstractPropertyEvaluator("UtilityFunction2") {
            public Object evaluate(IContextValueAccess context, IPropertyEvaluatorContext evalContext) {
                // evaluation code
                double utility = ...
                return new Double(utility);}});

        public Object evaluate(IContextValueAccess context, IPropertyEvaluatorContext evalContext) {
            // Default Utility (Downward-compatibility)
            return utilities.get("function1").evaluate(context, evalContext);
        }
    }
}

Figure 13.6 shows the definition of two static utility functions in the HelloWorldApp application. UtilityFunction1 is defined as default and will be taken if the user has not set any preferences. Using the adaptation middleware’s user interface as presented in Section 11.1.2, users can select which function should be used for the evaluation of application variants. Listing 13.3 shows the implementation (skeleton) of the two utility functions. The evaluate method in line 30 returns the default utility function and can be used by the original MUSIC middleware which does not support multiple functions.

The utility building and switching demonstrator is independent of the applied adaptation reasoning algorithm. Any of the original MUSIC algorithms or the new algorithms for user participation can be used.

13.5 Demonstrator E: Fuzzy-based Adaptation

The demonstrator for fuzzy-based adaptation is very similar to Demonstrator D on dynamic utility functions as we also have to define the required meta data. With the user interface of the adaptation middleware, the user can specify rules that determine
Figure 13.6: Demonstrator D: Specification of multiple static utility functions defining two oppositional behaviours.

the value of fuzzy metrics during inference (cf. Section 11.2.1). The antecedents of these rules are context dependencies, while the consequences are the aforementioned fuzzy metrics. Both context dependencies and fuzzy metrics have to be fuzzified first. In Figure 11.5 we have already shown the fuzzification of the plan property PowerConsumption and the context dependency BatteryLoad. We re-named the plan property to FuzzyPowerConsumption to clearly indicate its fuzzy background. The constant value of the plan property still holds (e.g. PowerConsumption=0.5 for the AppLogic1 realisation), but with the fuzzification process and the creation of rules from the middleware’s user interface, a new context information FuzzyPowerConsumption is created. This context information will be used by the utility function and defines how high the power consumption should be according to the user-defined rules. This time, we decided to keep LandscapeProvided and Landscape without any fuzzification. Providing the correct text orientation depending on the screen orientation is reasonable and fuzzification and rule definitions would make things unnecessary complicated.

Figure 13.7: Demonstrator E: Specification of the utility function with fuzzy metrics.

With the fuzzified plan properties, the context dependencies, and the fuzzy metric context information, we can set-up the utility function. Figure 13.7 illustrates the modified utility function including the FuzzyPowerConsumption property. Listing 13.4 lists the implemented pseudo code of this utility function. Landscape and LandscapeProvided are compared as before and define the utility value by 0.5 at maximum. In the next step, FuzzyPowerConsumption and PowerConsumption are compared using the proportional evaluation type and a linear calculation for this evaluation. This expression may add another 0.5 to the total utility value. In the best case, the landscape and the power
Listing 13.4: Demonstrator E: A utility function using fuzzy metrics and plan properties.

class UtilityFunction extends AbstractPropertyEvaluator {
    private static final long serialVersionUID = 1276002092723L;

    public Object evaluate(IContextValueAccess context,
                          IPropertyEvaluatorContext evalContext) {
        double utility = 0.0;

        boolean landscape = context.getBoolValue("#Thing.Concept.Entity.Device|this;
"+#Thing.Concept.Scope.Resource.Screen.Orientation", false);

        boolean landscapeProvided = ((Boolean) evalContext.evaluate("LandscapeProvided", context)).booleanValue();

        utility = landscape == landscapeProvided ? 0.5 : 0.0;

        double fuzzyPowerConsumption = context.getDoubleValue("#Thing.Concept.Entity.Middleware|self;
"+#Thing.Concept.Scope.Fuzzy.Reasoner;"+#HelloWorldApp|FuzzyPowerConsumption", 100.0);

        double powerConsumption = ((Double) evalContext.evaluate("PowerConsumption", context)).doubleValue();

        double eval = 100 - Math.abs(fuzzyPowerConsumption - powerConsumption) + Double.MIN_VALUE;

        utility += eval / 200;

        return utility;
    }
}

consumption evaluations reach their maximum value and the total utility will be 1.0 for this application variant.

The demonstrator for fuzzy-based adaptation is independent of the applied adaptation reasoning algorithm. Any of the original MUSIC algorithms or the new algorithms for user participation can be used because our solution builds on top of the original utility function reasoning mechanism.

13.6 Demonstrator F: Learning-based Adaptation

The demonstrator for learning-based adaptation reasoning makes use of the adaptation.plugins.learning adaptation algorithm which implements the learning approach presented in Section 11.3. The demonstrator does not require larger changes within the HelloWorldApp. However, for the training of application variants, the application must include components that allow communicating with the feedback manager of the adaptation middleware as in Demonstrator A (cf. Figure 13.1). It must be guaranteed that every application variant contains at least one feedback component. Furthermore, we have to define plans constituting a default application variant, so the adaptation knows which application variant it should instantiate when the knowledge base is empty.
or no suitable match could be found. Figure 13.8 depicts the specification of the default attribute for the realisation plans PortraitUI and AppLogic1. An application containing both components will have the highest default degree (cf. Section 10.4).

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PortraitUI</td>
<td>$\text{isDefault}=\text{true}$, $\text{PowerConsumption}=\text{500}$</td>
</tr>
<tr>
<td>AppLogic1</td>
<td>$\text{isDefault}=\text{true}$, $\text{JVMMemoryResourceService}=\text{30000}$</td>
</tr>
</tbody>
</table>

Figure 13.8: Demonstrator F: Specification of default components that constitute a possible default application variant after calculation of the default degree.

While we have not used any real user interface for plan preferences in Demonstrator A, we now require a graphical user interface for training the knowledge base. We extended the demonstrator’s OSGi bundle by a GUI based on the Swing framework (cf. Figure 13.9). The GUI will be instantiated by the SimpleHelloWorld, the PortraitUI, or the LandscapeUI realisation. With the GUI, we can activate and deactivate training mode, i.e. switch between live and operational mode. While in training mode, we can learn application variants by selecting the available plans and use manual adaptation to switch to an application variant containing the preferred plan(s). When switching to operational mode, the adaptation algorithm will select the best matching application variant based on the current situation and the rules specified via the middleware’s user interface (cf. Figure 11.7).

Figure 13.9: Demonstrator F: Sampling training data by preferring individual plans selected from the user interface.
Table 13.2: Required bundles for the extended MUSIC middleware with support for user participation. Mandatory bundles are highlighted bold.

<table>
<thead>
<tr>
<th>Bundle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adaptation</td>
<td>This bundle contains the adaptation middleware with our extensions Notification Manager, Feedback Manager, Preferences Manager, Learning Manager, Dynamic Utility Builder, and Fuzzy Reasoner. It requires one of the adaptations.plugins to be active.</td>
</tr>
<tr>
<td>adaptation.plugins.bruteforce</td>
<td>The default brute force adaptation reasoner should be included if no other reasoner wants to be used. It has no capabilities for user participation.</td>
</tr>
<tr>
<td>adaptation.plugins.manual</td>
<td>The brute force algorithm enhanced with capabilities to manage plan preferences for manual adaptation.</td>
</tr>
<tr>
<td>adaptation.plugins.userfocus</td>
<td>The user focus adaptation algorithm.</td>
</tr>
<tr>
<td>adaptation.plugins.learning</td>
<td>The learning-based adaptation algorithm.</td>
</tr>
<tr>
<td>adaptation</td>
<td>This bundle provides the communication infrastructure in terms of discovering and binding other middleware instances and services.</td>
</tr>
<tr>
<td>context</td>
<td>The context middleware. Requires context.plugins.</td>
</tr>
<tr>
<td>context.ontology</td>
<td>The bundle provides data structures to access the context ontology. Contains parts of the ontology.</td>
</tr>
<tr>
<td>context.repository.index</td>
<td>Responsible for storing and managing context data.</td>
</tr>
<tr>
<td>context.plugins.*</td>
<td>The different context plugins that deliver sensor information or calculated information to the context middleware. Select when required by application.</td>
</tr>
<tr>
<td>gui.swing</td>
<td>Provides the Swing-based middleware user interface (e.g. on Windows or Linux).</td>
</tr>
<tr>
<td>kernel</td>
<td>Framework classes.</td>
</tr>
<tr>
<td>manager</td>
<td>Management of OSGi bundles. e.g. (un-)installation.</td>
</tr>
<tr>
<td>model</td>
<td>Contains the information model, e.g. plan types, realisations, and properties.</td>
</tr>
<tr>
<td>model.extended</td>
<td>The extended model required for applications with user participation mechanisms.</td>
</tr>
<tr>
<td>negotiation</td>
<td>Negotiates Service Level Agreements (SLAs) with discovered or bound web services.</td>
</tr>
<tr>
<td>resources</td>
<td>Resource management, e.g. memory, or network.</td>
</tr>
<tr>
<td>resources.plugins.*</td>
<td>The plugins to retrieve resource information. Select when required by application. Enable context plugin master.resource.sensor if required as context plugins.</td>
</tr>
<tr>
<td>user</td>
<td>User management. Stores user preferences from adaptation middleware persistently.</td>
</tr>
</tbody>
</table>
14 Conclusion

Self-adaptive software promises intelligent anticipation by software in ubiquitous computing systems by adapting itself to changing run-time conditions. However, the research is focused on low-level adaptations and technical details of the adaptation approach. Ubiquitous computing systems typically include human involvement at some point, but the effects of adaptations to the humans were not considered adequately so far.

On the one hand we have seen thorough and comprehensive research on human factors in automated systems and on the other hand there is this field of self-adaptive software – emerged from automated systems – that does not consider the user sufficiently. During the past ten years, self-adaptive software has reached a level of maturity which requires a more extensive and multidisciplinary consideration within the different fields of application – especially from the socio-technical perspective in ubiquitous computing systems. This dissertation contributes to this socio-technical perspective on self-adaptive software. In the following, we summarise our research contributions and discuss key topics for future work.

14.1 Summary of Contributions

We claimed that user participation and the effects of adaptations to the user were not adequately addressed in the past. Although we think that almost every computing system has to interact with humans, there was still the possibility that researchers of self-adaptive software aim at scenarios with little or no user interaction. Hence, we conducted a literature study to analyse what type of application domains and example applications researchers used to motivate their work. In this study we analysed 41 publications including 38 example applications. In the results we have seen ubiquitous and mobile computing as typical application domains. Often, example applications are only provided as simple toy examples or textual descriptions. Evaluations were rarely conducted with real users. The results of the literature study supported our argumentation for more user participation in self-adaptive software.

In this work we made a significant effort to not only consider the claims of self-adaptive software, but also one of the major tradeoffs: the user finding himself out of the loop due to various sources of uncertainty within the development process, the environment, and the application itself. Our three-dimensional concept for user participation reintegrates the user back in the loop depending on the required degree of user participation. The three dimensions are *temporal participation*, *behavioural participation*, and *structural participation*. An orthogonal notification and feedback dimension serves as backbone.

The temporal participation dimension addresses ad-hoc and short-term interaction between the user and the self-adaptive system. With *manual adaptation* users have
mechanism to control the self-adaptive application whenever they want, e.g. if they are not content with the autonomous behaviour. Pre- and post-controllability were introduced as semi-autonomous adaptation mechanisms. Pre-controllability allows users to influence the adaptation behaviour before an adaptation happens whereas post-controllability allows the user to influence the adaptation after it has happened. Influence means that users can accept, deny, undo, or postpone adaptations. Interruptions by adaptations can distract the user from the current task. With user focuses we introduced special application components that cannot be autonomously adapted by an adaptation controller. Each user focus describes a collection of functions required to fulfil a particular task. While user focuses are a static concept applied at design time by developers, interaction activity detection analyses the user while interacting with the running application. From the deduced activity class we can conclude if a user should be interrupted by an adaptation or whether a specific notification should be sent instead. If users are not content with the autonomous behaviour at all, we allow pausing or stopping any self-adaptation processes. Default template reasoning determines an application variant if no other variant is suitable or if the self-adaptive behaviour is currently stopped.

Within the behavioural participation dimension we address the long-term and explicit modification of the application behaviour. We allow users to adjust the application’s behaviour to their needs and to make changes that are persistent over time. The concept of dynamic utility functions is an extension to the classic utility function policy used by adaptation reasoners. We simplify the creation of utility functions by splitting the overall function into multiple sub-functions addressing only a single quality metric. The dynamic utility functions can be created from the middleware’s user interface, even on smaller screens. Moreover, users can store their created utility functions as so-called behaviour profiles to address different situations properly. Run-time switching between these profiles allows a change in behaviour without developer intervention or re-deploying the application. The fuzzy-based adaptation reasoning eases the management and alteration of run-time behaviour. Rules and fuzzy metrics with their linguistic terms are defined in natural language and are easier to understand than cumbersome utility functions. The combination of fuzzy-based reasoning and utility functions eases understanding from a user’s perspective, but keeps the expressiveness and scalability of utility functions. Adaptive applications that automatically adjust to changing user habits are the motivation for learning-based adaptation reasoning. The learning-based reasoner works with predefined situations which in turn make use of the already integrated fuzzy system. With a k-nearest neighbour classification algorithm we compare the current situation according to the predefined situations and select the best suitable application variant, i.e. the variant with the least distance. Although this approach requires a separate training phase, the knowledge base will be adjusted by user feedback, too.

The notification and feedback concept is very simple, but probably the most basic improvement for user participation in self-adaptive software. Although it is clever from a technical perspective to have loosely coupled components that are instantiated at run-time via a reflection mechanism, it keeps the user out of the loop. By introducing bi-direction communication and resolving the loosely coupling slightly, we achieve more intelligibility and controllability. Without the notification and feedback concept, many of the introduced mechanism do not work. Although some of the mechanisms like the fuzzy-based or the learning-based adaptation reasoning are not completely new, they have however, not been applied yet to self-adaptive software to improve user participation.
The tradeoffs of self-adaptive software (like the impact on users) can only be analysed properly when conducting extensive user studies. The Meet-U case study served as an evaluation scenario for the notification and feedback concept in combination with pre- and post-adaptation controllability. We have seen that the different design variations providing controllability resulted in a much higher user acceptance than a design variation with a completely autonomous adaptation. However, there is no perfect design variation fitting all adaptations. The preferred design variation depends on the type of adaptation, the context of use, and personal preferences. More user studies are required to properly analyse the effects of adaptation to the user.

We applied and integrated the different mechanisms for user participation to the MUSIC middleware and MDD methodology which is a proved, efficient, and powerful solution for self-adaptation in mobile and ubiquitous computing environments. The MUSIC middleware incorporates a MAPE-K adaptation loop which is the predominant solution for application-level adaptation. Our systematic solution integrates the user in self-adaptive software by defining user interfaces to the different elements of the MAPE-K adaptation loop. However, we have to admit that the complexity in the development of such software increases when also considering the user. There is a need for a more holistic development methodology for self-adaptive software that respects the user already during the requirements analysis phase.

14.2 Outlook and Future Work

This dissertation is a first attempt to enable the systematic integration of the user in self-adaptive software. The proposed concepts for user participation are a significant contribution to the practical relevance of self-adaptive software. However, not all challenges could be addressed and new challenges arose during implementation and evaluation. We leave these following challenges as opportunities for future research:

- **Integration in other existing self-adaptive solutions**: We applied the different mechanisms for user participation to the MUSIC middleware and the model-driven development approach of the MUSIC development methodology. Although the MUSIC approach is comprehensive and general, it must be shown how well other self-adaptive solutions can be improved with the mechanisms for user participation. As the proposed concepts within the three dimensions are not specific to the MUSIC approach, we assume this will be only a technical and implementational matter.

- **Configuration of the interruption level**: We discussed interruptions caused by adaptations in Chapter 2. The notification types presented in Chapter 9 cause different degrees of interruption, too. However, user attention is a limited resource in human-computer interaction. Within the Meet-U case study it turned out that users prefer different types of notifications for different adaptations. Instead of manually defining the notification type, users could specify the level of interruption for the current situation or specific adaptations. In this case, the notification manager will decide what type of notification to send.

- **Optimisation of the fuzzy-based adaptation reasoning**: We have shown the feasibility of a fuzzy-based adaptation reasoning process. However, the creation of rules is still tricky and the current approach has some limitations regarding
the implementation. We propose the following improvements: first, implement support for OR conjugations of fuzzy variables. Second, implement support for hierarchical fuzzy rules [167] which will minimise the total number of rules. And third, implement a debugging and testing tool that helps expert users or developers to define a suitable set of rules by simulating inference.

- **Support for multiple control loops:** So far we considered the MAPE-K adaptation loop which uses context information and information on the self-performance to adapt the software. This means we have a central instance controlling the adaptation behaviour which also means the integration of user participation is probably easier than with multiple distributed control loops. It has to be analysed in how far the proposed mechanism for user participation can be applied to multiple control loops, too.

- **Improvement of intelligibility:** Although intelligibility was not in the focus of our research, some of the presented concepts for user participation create intelligibility implicitly. For example, the notification concept in combination with pre- and post-adaptation controllability reveals adaptation actions to the user. This high-level information can help users to create a more accurate mental model. We think that revealing low-level context information or quality parameters is not very helpful for ordinary end-users, but it should be evaluated if explicit high-level explanations on adaptation behaviour are helpful, too.

- **End-to-end development process:** The evaluation within the Meet-U case study revealed that there is no appropriate adaptation notification design that fits every adaptation. But even with a clear recommendation, developers have to develop every adaptation individually. Hence, we claim for an end-to-end development process that starts with an adaptation-oriented requirements analysis and ends with testing capabilities for adaptations with user participation. Different types of adaptation have to be identified during the requirements analysis. Then, adaptation patterns should be applied to derive appropriate user participation concepts.

- **Comprehensive user-based evaluation:** The evaluation study with the Meet-U application covered the notification and feedback concept with pre- and post-controllability in regard to user acceptance. The study was very complex and involved more than nine month of planning, execution, and analysis. Similar studies must be done for the other user participation mechanisms not evaluated in this work, especially for the learning-based approach. There is no doubt whether these mechanisms facilitate user participation, but the question is how well they do it. Further, we need to know when to use dynamic utility functions or when we should prefer the learning-based over the fuzzy-based reasoning approach.

- **Structural dimension for user participation:** In Chapter 7 we present three dimensions on how user participation can be achieved in self-adaptive software. So far we have addressed the temporal and the behavioural dimension. We leave the structural dimension for future work. On the one hand, the structural dimension addresses the field of end-user programming in which users can contribute their own software artefacts. On the other hand, it allows users to control the selection of components and services that are used by an application. Both are important and interesting ways for user participation.
Part IV

Appendices
A Fuzzy Reasoning Example

FUNCTION_BLOCK heating_example

// Define input variables
VAR_INPUT
    temp : REAL;
    time : REAL;
END_VAR

// Define output variable
VAR_OUTPUT
    heating : REAL;
END_VAR

// Fuzzify input variable 'temp'
FUZZIFY temp
    TERM cold := (0,1) (10,1) (20,0);
    TERM normal := (15, 0) (20, 1) (25,0);
    TERM hot := (20, 0) (30, 1) (40, 1);
END_FUZZIFY

// Fuzzify input variable 'time'
FUZZIFY time
    TERM morning := (5, 0) (7, 1) (10, 1) (12, 0);
    TERM day := (7, 0) (8,1) (19,1) (20,0);
    TERM evening := (17, 0) (18, 1) (21, 1)(23, 1);
END_FUZZIFY

// Defuzzify output variable 'heating'
DEFUZZIFY heating
    TERM off := (0,1) (1,0);
    TERM low := (1,0) (10,1) (30,1) (40,0);
    TERM normal := (20,0) (40,1) (60,1) (80,0);
    TERM high := (70,0) (80,1) (100,1);
    // Use 'Center Of Gravity' defuzzification method
    METHOD : COG;
    // Default value is 0 (if no rule activates defuzzifier)
    DEFAULT := 0;
END_DEFUZZIFY

RULEBLOCK No1
    // Use 'min' for 'and' (also implicitly use 'max' for 'or' to fulfil DeMorgan's Law)
    AND : MIN;
    // Use 'min' activation method
    ACT : MIN;
    // Use 'max' accumulation method
    ACCU : MAX;

    RULE 1 : IF temp IS normal THEN heating IS normal;
    RULE 2 : IF temp IS cold THEN heating IS high;
    RULE 3 : IF temp IS normal AND time IS evening THEN heating IS off;
    RULE 4 : IF temp IS hot THEN heating IS off;
    RULE 5 : IF time IS NOT day THEN heating IS low;
    RULE 6 : IF time IS morning THEN heating IS high;
END_RULEBLOCK

END_FUNCTION_BLOCK

Listing A.1: FCL function block for the heating example in Chapter 6.
B Provided Examples for Literature Study on Self-Adaptive Applications

Table B.1: Application examples provided by publications on solutions for self-adaptation. Some of the publications provided multiple examples. See Chapter 4 for details on the literature study.

<table>
<thead>
<tr>
<th>#</th>
<th>Publication</th>
<th>Research Domain</th>
<th>Application Domain</th>
<th>Maturity of Example</th>
<th>User Impact</th>
<th>Type of Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alia et al. [172]</td>
<td>ubiquitous</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>2</td>
<td>Almeida et al. [173]</td>
<td>ubiquitous</td>
<td>smart home</td>
<td>prototype</td>
<td>high</td>
<td>none</td>
</tr>
<tr>
<td>3</td>
<td>Anthony [174]</td>
<td>none</td>
<td>stock trading</td>
<td>textual</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>4</td>
<td>Baude et al. [175]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>5</td>
<td>Beggas et al. [176]</td>
<td>mobile</td>
<td>none</td>
<td>high</td>
<td>claims</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Bencomo et al. [177]</td>
<td>service-based</td>
<td>catastrophe</td>
<td>prototype</td>
<td>low</td>
<td>none</td>
</tr>
<tr>
<td>7</td>
<td>Bratskas et al. [178]</td>
<td>ubiquitous</td>
<td>smart home</td>
<td>early prototype</td>
<td>high</td>
<td>proof of concept</td>
</tr>
<tr>
<td>8</td>
<td>Capra et al. [179]</td>
<td>mobile</td>
<td>social</td>
<td>prototype</td>
<td>high</td>
<td>claims</td>
</tr>
<tr>
<td>9</td>
<td>Cardoso et al. [180]</td>
<td>none</td>
<td>media</td>
<td>prototype</td>
<td>low</td>
<td>proof of concept</td>
</tr>
<tr>
<td>10</td>
<td>Chaari et al. [181]</td>
<td>service-based</td>
<td>healthcare</td>
<td>prototype</td>
<td>high</td>
<td>none</td>
</tr>
<tr>
<td>11</td>
<td>Chefrour [182]</td>
<td>mobile</td>
<td>media</td>
<td>textual</td>
<td>low</td>
<td>none</td>
</tr>
<tr>
<td>12</td>
<td>Cheng et al. [183]</td>
<td>service-based</td>
<td>information</td>
<td>early prototype</td>
<td>high</td>
<td>proof of concept</td>
</tr>
<tr>
<td>13</td>
<td>Chuang and Chan [184]</td>
<td>mobile</td>
<td>media</td>
<td>prototype</td>
<td>medium</td>
<td>claims</td>
</tr>
<tr>
<td>14</td>
<td>Chun et al. [185]</td>
<td>ubiquitous</td>
<td>smart home</td>
<td>textual</td>
<td>high</td>
<td>none</td>
</tr>
<tr>
<td>15</td>
<td>Esfahani et al. [186]</td>
<td>service-based</td>
<td>travel</td>
<td>prototype</td>
<td>high</td>
<td>claims</td>
</tr>
<tr>
<td>16</td>
<td>Floch et al. [187]</td>
<td>mobile</td>
<td>technician support</td>
<td>prototype</td>
<td>high</td>
<td>claims</td>
</tr>
<tr>
<td>17</td>
<td>Floch et al. [187]</td>
<td>mobile</td>
<td>travel</td>
<td>prototype</td>
<td>high</td>
<td>claims</td>
</tr>
<tr>
<td>18</td>
<td>Floch et al. [187]</td>
<td>mobile</td>
<td>healthcare</td>
<td>prototype</td>
<td>high</td>
<td>claims</td>
</tr>
<tr>
<td>19</td>
<td>Floch et al. [187]</td>
<td>mobile</td>
<td>healthcare</td>
<td>prototype</td>
<td>high</td>
<td>claims</td>
</tr>
<tr>
<td>20</td>
<td>Floch et al. [187]</td>
<td>mobile</td>
<td>travel</td>
<td>prototype</td>
<td>high</td>
<td>claims</td>
</tr>
<tr>
<td>21</td>
<td>Floch et al. [187]</td>
<td>mobile</td>
<td>social</td>
<td>prototype</td>
<td>high</td>
<td>claims</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>#</th>
<th>Publication</th>
<th>Research Domain</th>
<th>Application Domain</th>
<th>Maturity of Example</th>
<th>User Impact</th>
<th>Type of Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Garlan et al. [188]</td>
<td>service-based systems</td>
<td>information</td>
<td>prototype</td>
<td>low</td>
<td>claims</td>
</tr>
<tr>
<td>23</td>
<td>Garlan et al. [188]</td>
<td>service-based systems</td>
<td>media</td>
<td>prototype</td>
<td>high</td>
<td>none</td>
</tr>
<tr>
<td>24</td>
<td>Geihs et al. [189]</td>
<td>mobile computing</td>
<td>technician support</td>
<td>prototype</td>
<td>high</td>
<td>claims</td>
</tr>
<tr>
<td>25</td>
<td>Geihs et al. [189]</td>
<td>mobile computing</td>
<td>technician support</td>
<td>prototype</td>
<td>high</td>
<td>claims</td>
</tr>
<tr>
<td>26</td>
<td>Georfas and Taylor [190]</td>
<td>robotics</td>
<td>none</td>
<td>textual</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>27</td>
<td>Georfas and Taylor [190]</td>
<td>robotics</td>
<td>none</td>
<td>textual</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>28</td>
<td>Huebscher and McCann [191]</td>
<td>service-based systems</td>
<td>smart home</td>
<td>textual</td>
<td>high</td>
<td>none</td>
</tr>
<tr>
<td>29</td>
<td>Keeney and Cahill [192]</td>
<td>mobile computing</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>30</td>
<td>Khan [193]</td>
<td>mobile computing</td>
<td>travel</td>
<td>textual</td>
<td>high</td>
<td>claims</td>
</tr>
<tr>
<td>31</td>
<td>Kramer and Magee [194]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>32</td>
<td>Liu [195]</td>
<td>grid computing</td>
<td>catastrophe</td>
<td>prototype</td>
<td>medium</td>
<td>claims</td>
</tr>
<tr>
<td>33</td>
<td>Lopes et al. [196]</td>
<td>ubiquitous computing</td>
<td>healthcare</td>
<td>prototype</td>
<td>high</td>
<td>tradeoffs</td>
</tr>
<tr>
<td>34</td>
<td>Malek et al. [197]</td>
<td>robotics</td>
<td>robotics</td>
<td>prototype</td>
<td>high</td>
<td>claims</td>
</tr>
<tr>
<td>35</td>
<td>Menasce et al. [198]</td>
<td>service-based systems</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>36</td>
<td>Oreizy et al. [199]</td>
<td>robotics</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>37</td>
<td>Pascual et al. [200]</td>
<td>mobile computing</td>
<td>social</td>
<td>prototype</td>
<td>high</td>
<td>claims</td>
</tr>
<tr>
<td>38</td>
<td>Paspallis and Papadopoulos [201]</td>
<td>mobile computing</td>
<td>media</td>
<td>prototype</td>
<td>high</td>
<td>none</td>
</tr>
<tr>
<td>39</td>
<td>Pinto et al. [202]</td>
<td>robotics</td>
<td>science</td>
<td>prototype</td>
<td>high</td>
<td>proof of concept</td>
</tr>
<tr>
<td>40</td>
<td>Ramaswamy et al. [203]</td>
<td>robotics</td>
<td>none</td>
<td>none</td>
<td>high</td>
<td>none</td>
</tr>
<tr>
<td>41</td>
<td>Rasche et al. [204]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>42</td>
<td>Romero et al. [205]</td>
<td>mobile computing</td>
<td>ecomerce</td>
<td>textual</td>
<td>high</td>
<td>none</td>
</tr>
<tr>
<td>43</td>
<td>Sallem and Silva e Silva [206]</td>
<td>grid computing</td>
<td>none</td>
<td>prototype</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>44</td>
<td>Shang-Wen Cheng and Garlan [207]</td>
<td>service-based systems</td>
<td>information</td>
<td>prototype</td>
<td>high</td>
<td>tradeoffs</td>
</tr>
<tr>
<td>45</td>
<td>Sousa et al. [208]</td>
<td>ubiquitous computing</td>
<td>media</td>
<td>prototype</td>
<td>high</td>
<td>claims</td>
</tr>
<tr>
<td>46</td>
<td>Toninelli et al. [209]</td>
<td>mobile computing</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>47</td>
<td>Yamin et al. [210]</td>
<td>ubiquitous computing</td>
<td>media</td>
<td>textual</td>
<td>high</td>
<td>none</td>
</tr>
<tr>
<td>48</td>
<td>Yamin et al. [210]</td>
<td>ubiquitous computing</td>
<td>science</td>
<td>textual</td>
<td>low</td>
<td>none</td>
</tr>
</tbody>
</table>

190 Provided Examples for Literature Study on Self-Adaptive Applications
C MUSIC

C.1 Taxonomy

Figure C.1: Taxonomy of the MUSIC adaptation approach (cf. Figure 3.2).
C.2 Extended UML Modelling Profile

(a) Types.

(b) Realisations.

Figure C.2: The extended MUSIC UML modelling profile with support for user participation.
D Application Models

D.1 Meet-U

Figure D.1: Two composite realisations of the Meet-U application. For the composite realisations Planning and AtEvent see Section 12.1.
Figure D.2: Component type realisation of the Meet-U application.

Figure D.3: Parameter settings for parametric adaptation in the Meet-U application.

194 Application Models
Figure D.4: Variability model with property type definition for the HelloWorldApp application.
E Bibliographies

E.1 Bibliography


[169] Qian Yang, Danfeng Yao, James Garnett, and Kaitlyn Muller. Using a Trust Inference Model for Flexible and Controlled Information Sharing During


E.2 Publications Used for Literature Study


[178] Pyrros Bratskas, Nearchos Paspallis, Konstantinos Kakousis, and George Angelos Papadopoulos. *Applying Utility Functions to Adaptation Planning for Home


[197] Sam Malek, George Edwards, Yurii Brun, Hossein Tajalli, Joshua Garcia, Ivo Krka, Nenad Medvidovic, Marija Mikic-Rakic, and Gaurav S. Sukhatme. **An


E.3 Publications as (Co-)Author


Kassel, im März 2014

M. Sc. Christoph Evers