4. Domain Engineering of Ubiquitous Applications

"One significant aspect of this emerging mode of computing is the constantly changing execution environment… Similarly, the computer user may move from one location to another, joining and leaving groups of people, and frequently interacting with computers while in changing social situations".

Bill N. Schilit, Norman Adams, and Roy Want, "Context-Aware Computing Applications," IEEE Workshop on Mobile Computing Systems and Applications, 1994.

In this Chapter, we describe the technological support we developed to make our approach viable as well as our reuse-oriented support sets – i.e. building blocks. Figure 4.1 shows the transversal domains of Ubiquitous Computing and Intentional MAS. Based on these domains, we developed the building blocks in the *Domain Engineering of Ubiquitous Applications*.

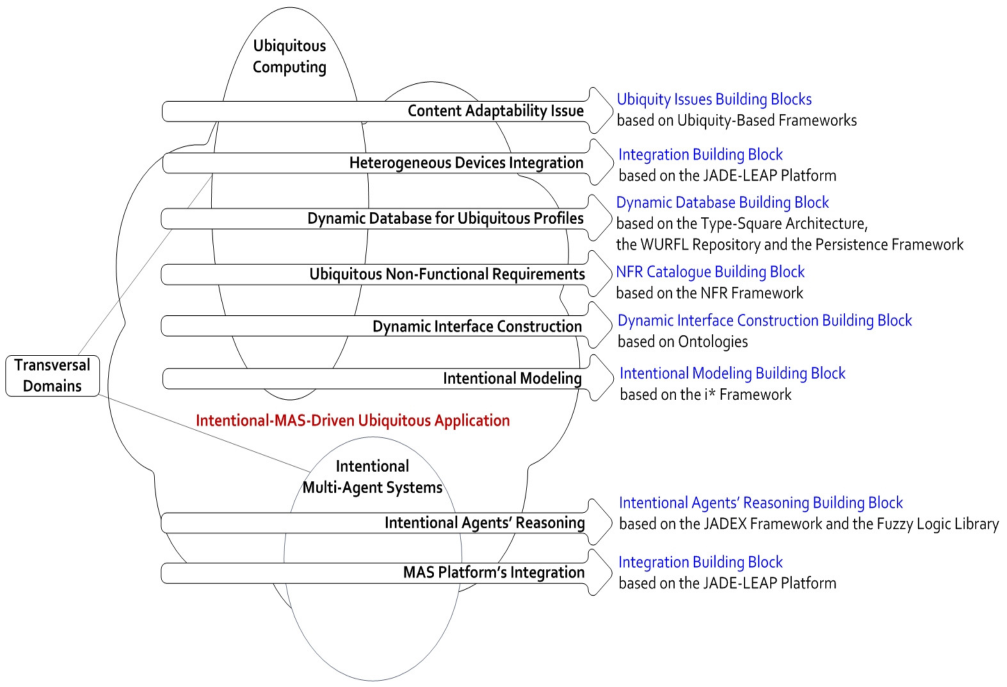

Ubiquitous Computing
Content Adaptability Issue
Heterogeneous Devices Integration
Dynamic Database for Ubiquitous Profiles
Ubiquitous Non-Functional Requirements
Dynamic Interface Construction
Intentional Modeling
Transversal Domains
Intentional-MAS-Driven Ubiquitous Application
Intentional Multi-Agent Systems
Intentional Agents' Reasoning
MAS Platform's Integration
Ubiquity Issues Building Blocks based on Ubiquity-Based Frameworks
Integration Building Block based on the JADE-LEAP Platform
Dynamic Database Building Block based on the Type-Square Architecture, the WURFL Repository and the Persistence Framework
NFR Catalogue Building Block based on the NFR Framework
Dynamic Interface Construction Building Block based on Ontologies
Intentional Modeling Building Block based on the i* Framework
Intentional Agents' Reasoning Building Block based on the JADEX Framework and the Fuzzy Logic Library
Integration Building Block based on the JADE-LEAP Platform

Figure 4.1 - Reusable *Building Blocks* for intentional ubiquitous applications

We focus our efforts on the *Development for Reuse* based on the main ubiquitous concerns/issues and the intentional MAS paradigm. Since 2007, we have been developing ubiquitous applications in different cognitive domains (e.g. e-commerce and dental clinic) (Serrano et al. 2011a; Serrano and Lucena 2011a; Serrano and Lucena 2011b; Serrano and Lucena 2011c; Serrano and Lucena 2010a; Serrano and Lucena 2010b; Serrano et al. 2009; Serrano et al. 2008). We started the development of these applications by using behavioral agents. However, we tried to improve the cognitive capacity of the agents in ever-changing environments by using the intentionality abstraction from the Goal-Orientation paradigm. Therefore, we developed some of those applications centered on intentional MAS. According to our experimental work and the literature, there are some advantages in developing intentional-agents, such as the BDI-based agents presented in our reuse-oriented building blocks as well as other goal-based agents. According to (Dignum and Conte 1997), the new goals formation is a fundamental feature of autonomous entities, *"existing formal theories of agents are found essentially inadequate to account for the formation of new goals and intentions of the agent"*. The agent's cognition capacity and the rationale significantly increase using the distributed intentionality (Yu 1997) as a goal-orientation-centered approach. The agents based their decisions on reasoning techniques and the user satisfaction, being aware of the ubiquitous context. In this scenario, the context awareness is centered on the agents' beliefs, desires and intentions as interpretation of the human-mental states. In addition, some common problems are avoided by using BDI-based agents, for example: it is really simple to deal with the agents' adaptability according to different ever-changing environments, by dynamically updating the agents' knowledge bases, their beliefs, their goals' formation and their sequence of tasks to achieve the desired goals. Strengthening our argumentation, we can define the world "intention" as *the state of one's mind at the time one carries out an action.*

Our experimental work helped us to compose an adequate view of the main ubiquitous concerns/issues and also to provide a suitable approach centered on the development for reuse to systematically and incrementally construct intentional-MAS-driven ubiquitous applications. Among different concerns/issues that our reuse-oriented approach deals with, we have: (i) the intentional modeling of ubiquitous applications; (ii) the non-functional requirements (NFRs) modeling by considering their interdependencies and operationalizations; (iii) the integration of heterogeneous devices as well as the integration of the MAS platform composed of containers that represent distributed environments in ubiquitous contexts; (iv) the intentional agents' reasoning by considering the goals' formation, ever-changing situations (e.g. users with different preferences, intrinsic mobility and devices in constant evolution), quality criteria (e.g. security, response time, performance and others), privacy & personalization balancing, invisibility & transparency balancing, and agents' inter-operability and communication need; (v) the interface construction at runtime by adapting contents (e.g. images, videos, files and texts) and different Graphical User Interface (GUI) elements according to the devices features and other ubiquitous profiles information; (vi) the ubiquitous specific issues (e.g. content adaptability and context awareness); and (vii) the ubiquitous profiles manipulation (e.g. store, retrieving and update) "on the fly."

Thus, we zoom in the *Intentional Modeling Building Block* focused on the i* Framework; the *NFR Catalogue Building Block* based on the NFR Framework; the *Integration Building Block* centered on the JADE-LEAP Platform; the *Intentional Agents' Reasoning Building Block* based on the JADEX Framework and the Fuzzy-Logic Library; the *Dynamic Interface Construction Building Block* based on Ontologies; the *Ubiquity Issues Building Blocks* focused on Ubiquity-Based Frameworks; and the *Dynamic Database Building Block* centered on the Type-Square Architecture, the WURFL Repository and the Persistence Framework. The use of frameworks, libraries, patterns and models to support the software development process has grown in the last few years. Contributing to this field, the proposed approach – as mentioned – also uses those resources to promote the reuse in ubiquitous contexts. Finally, Section 4.8 summarizes the Chapter by presenting some concluding remarks.

4.1. Intentional Modeling Building Block

We propose the use of the intentionality abstraction to model ubiquitous applications based on the stakeholders' beliefs, desires and intentions in order to improve the specification/documentation/modeling of the human practical reasoning (Bratman 1999) in ever-changing contexts.

The i* Framework (or iStar, which means *Distributed Intentionality*) is an initiative of the **U**niversity of **T**oronto (UofT) in order to model applications centered on the **G**oal-**O**riented **R**equirements **E**ngineering (GORE) (Mylopoulos 2008). It proposes an agent-oriented approach to requirements modeling focused on the intentional agents' properties, such as goals, beliefs, abilities and commitments. Therefore, the i* Framework offers two models: (i) the **S**trategic-**D**ependency (SD) model; and (ii) the **S**trategic-**R**ationale (SR) model. The former support is used to model the dependencies between actors/actors, actors/agents and agents/agents by, for example, giving rise to opportunities and vulnerabilities. The latter support models tasks as alternative actions to achieve the actor's and/or the agent's goals by assessing her/his/its strategic positioning in a specific context.

The intentionality-based modeling is particularly interesting for dealing with non-functional requirements (e.g. dependability, accountability and security), called softgoals in the i*. The i* models offer resources to establish the impacts between the alternatives (i.e. i* tasks to achieve functional requirements – i.e. i* goals) and the softgoals of the application. Figure 4.2 – designed by using the OME tool (OME 2011) (i.e. a specific tool to model applications according to the i* Framework) – shows a simple situation, in which the i* abstractions are used to model two alternative actions (i.e. *Perform the Patient's Registration in a Dental Clinic With a Privacy-Aware Application* and *Perform the Patient's Registration in a Dental Clinic Without a Privacy-Aware Application*) to achieve the patient's goal – *Patient's Registration Be Performed.* The dependencies between the *Patient* actor and the *Attendant* agent are also presented.

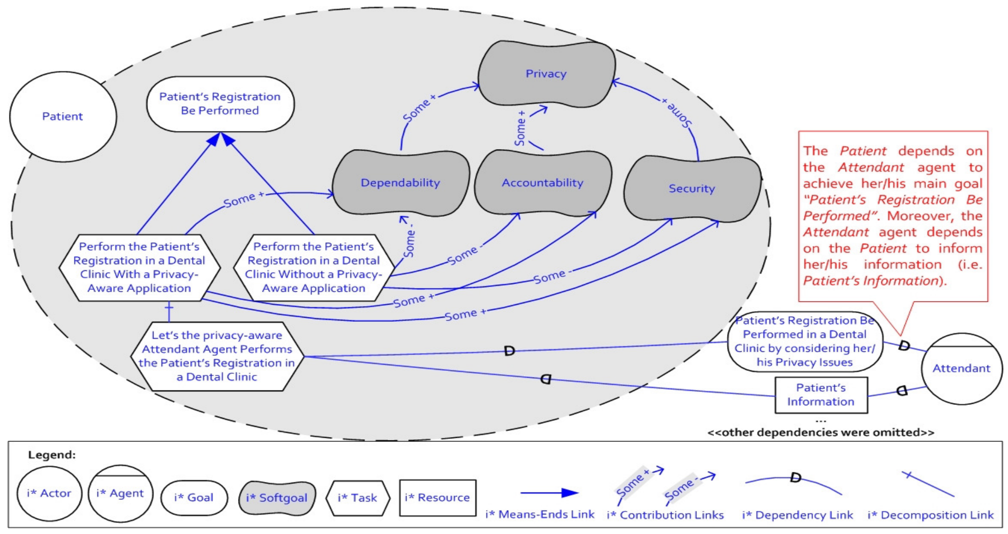

Patient
Patient's Registration Be Performed
Privacy
Some +
Some +
Some +
Dependability
Accountability
Security
Some -
Some -
Some -
Some +
Some +
Perform the Patient's Registration in a Dental Clinic With a Privacy-Aware Application
Perform the Patient's Registration in a Dental Clinic Without a Privacy-Aware Application
Let's the privacy-aware Attendant Agent Performs the Patient's Registration in a Dental Clinic
The Patient depends on the Attendant agent to achieve her/his main goal "Patient's Registration Be Performed". Moreover, the Attendant agent depends on the Patient to inform her/his information (i.e. Patient's Information).
Patient's Registration Be Performed in a Dental Clinic by considering her/his Privacy Issues
D
D
D
D
Attendant
Patient's Information
<<other dependencies were omitted>>
Legend:
i* Actor
i* Agent
i* Goal
i* Softgoal
i* Task
i* Resource
i* Means-Ends Link
Some +
Some -
i* Contribution Links
D
i* Dependency Link
i* Decomposition Link

Figure 4.2 - Example of i* model based on the dental clinic domain

Moreover, the model also represents the contributions between the alternatives and the softgoals – *e.g. Perform the Patient's Registration in a Dental Clinic with a Privacy-Aware Application* positively contributes (*some* +) to the *Dependability* and *Perform the Patient's Registration in a Dental Clinic without a Privacy-Aware Application* negatively contributes (*some* -) to the *Dependability.* Furthermore, we represent the impacts of the softgoals *Dependability*, *Accountability* and *Security* to the main softgoal *Privacy* – e.g. all of these softgoals positively contribute (*some* +) to *Privacy*.

Based on the intentionality abstraction and the i* Framework models, we propose a building block – called *Intentional Modeling Building Block* (Figure 4.3) – to model ubiquitous applications by considering the intrinsic ever-changing contexts they are embedded; the heterogeneity of the users' preferences, the necessity to deal with different quality criteria (i.e. non-functional requirements), and other ubiquitous issues. Therefore, this building block is mainly composed of the i* Framework conceptual model; the association between abstractions of intentional ubiquitous applications and i* abstractions, which is briefly illustrated in Figure 4.4; and different ubiquitous design patterns modeled by using intentionality. These ubiquitous design patterns were first proposed by (Landay and Borriello 2003) and some of them modeled with the i* abstractions by our research group. The main idea is to facilitate the model reuse as well as the proliferation of good practices in the incremental and systematic development of ubiquitous applications. Figure 4.5 briefly shows the Find-a-Friend ubiquitous design pattern by considering its background, solution and some more details.

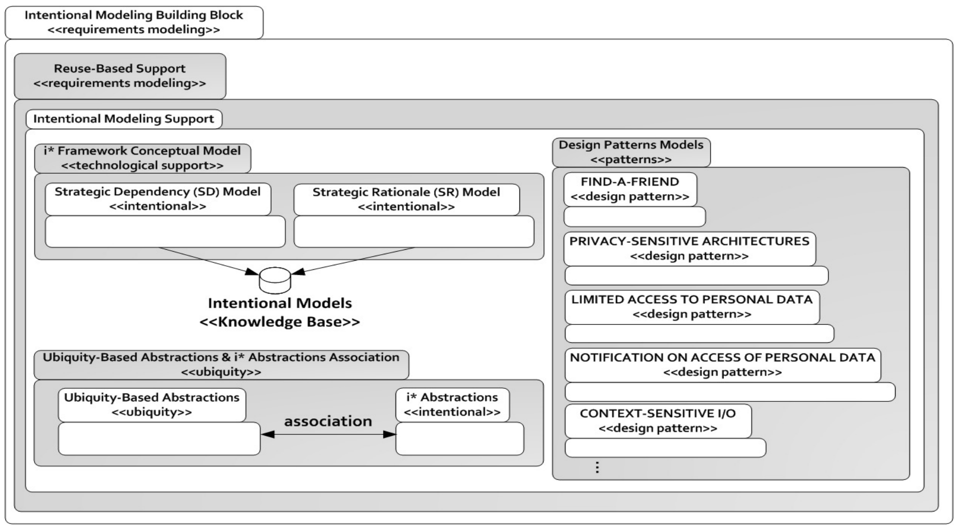

Intentional Modeling Building Block
<<requirements modeling>>
Reuse-Based Support
<<requirements modeling>>
Intentional Modeling Support
i* Framework Conceptual Model
<<technological support>>
Strategic Dependency (SD) Model
<<intentional>>
Strategic Rationale (SR) Model
<<intentional>>
Intentional Models
<<Knowledge Base>>
Ubiquity-Based Abstractions & i* Abstractions Association
<<ubiquity>>
Ubiquity-Based Abstractions
<<ubiquity>>
association
i* Abstractions
<<intentional>>
Design Patterns Models
<<patterns>>
FIND-A-FRIEND
<<design pattern>>
PRIVACY-SENSITIVE ARCHITECTURES
<<design pattern>>
LIMITED ACCESS TO PERSONAL DATA
<<design pattern>>
NOTIFICATION ON ACCESS OF PERSONAL DATA
<<design pattern>>
CONTEXT-SENSITIVE I/O
<<design pattern>>
⋮

Figure 4.3 - *Intentional Modeling Building Block* packages

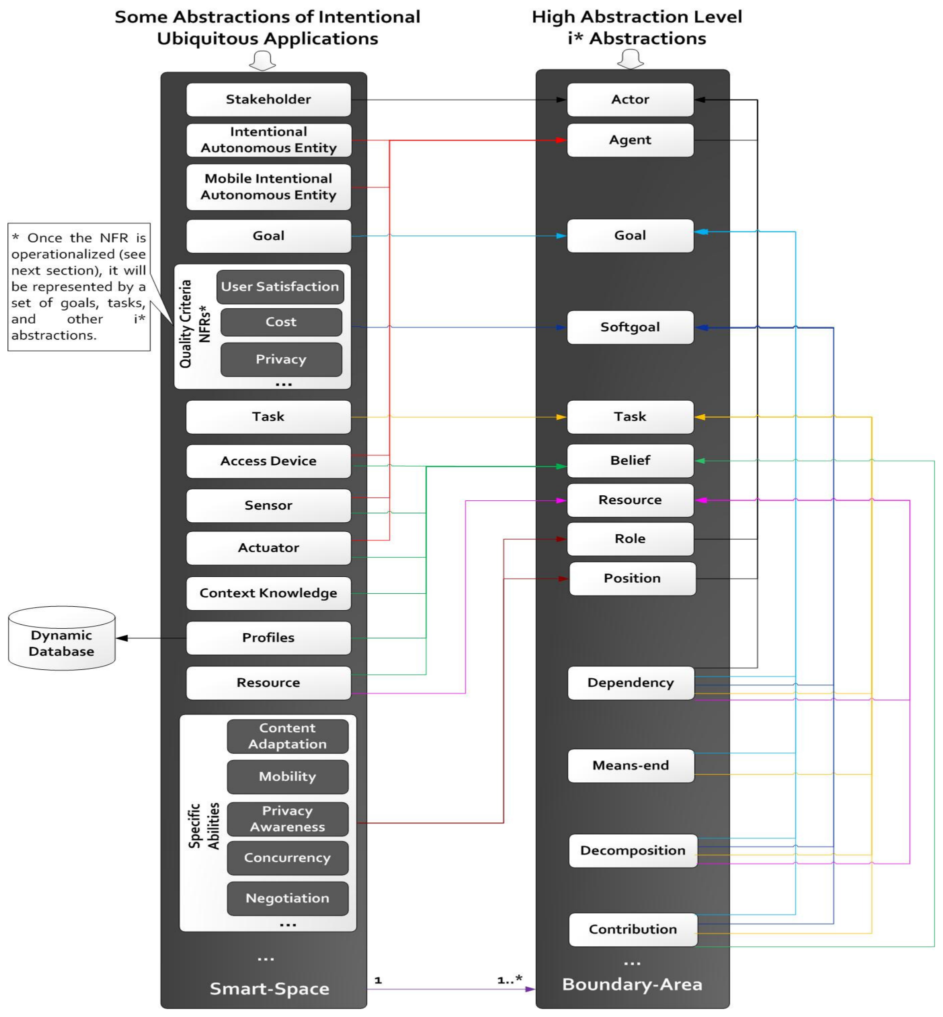
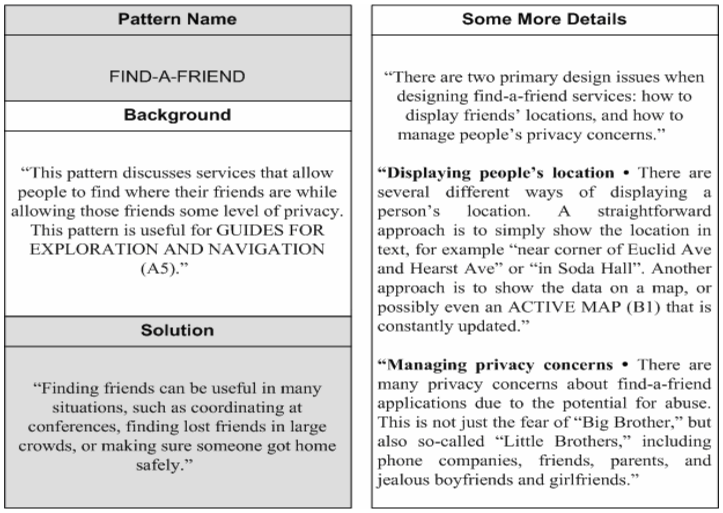


Figure 4.4 - Association between abstractions of intentional-MAS-driven ubiquitous applications and i* abstractions[1]



Pattern Name	Some More Details
FIND-A-FRIEND	"There are two primary design issues when designing find-a-friend services: how to display friends' locations, and how to manage people's privacy concerns." **"Displaying people's location •** There are several different ways of displaying a person's location. A straightforward approach is to simply show the location in text, for example "near corner of Euclid Ave and Hearst Ave" or "in Soda Hall". Another approach is to show the data on a map, or possibly even an ACTIVE MAP (B1) that is constantly updated." **"Managing privacy concerns •** There are many privacy concerns about find-a-friend applications due to the potential for abuse. This is not just the fear of "Big Brother," but also so-called "Little Brothers," including phone companies, friends, parents, and jealous boyfriends and girlfriends."
Background	
"This pattern discusses services that allow people to find where their friends are while allowing those friends some level of privacy. This pattern is useful for GUIDES FOR EXPLORATION AND NAVIGATION (A5)."	
Solution	
"Finding friends can be useful in many situations, such as coordinating at conferences, finding lost friends in large crowds, or making sure someone got home safely."	

Figure 4.5 - Find-a-Friend ubiquitous design pattern (adapted from (Landay and Borriello 2003))

[1] Some associations were omitted in order to facilitate the visualization.

4.2.
NFR Catalogue Building Block

In order to specifically deal with non-functional requirements commonly found in ubiquitous applications, we propose a catalogue centered on the NFR Framework. The NFR Framework constitutes a **G**oal-**O**riented **R**equirements **E**ngineering (hereafter GORE) approach to capturing NFRs in the domain of interest, and defining their interdependencies and operationalizations. Nowadays, there is much interest in this kind of approach within the Requirements Engineering community as goal-oriented elaboration processes end where traditional ones (e.g. RUP and other object-oriented approaches) begin. Thus, the NFR Framework focuses on activities that precede the requirements specification, and results in models that can be used during the design stage to drive and validate architectural decisions.

We chose the NFR Framework because it allows for capturing alternatives for different NFRs; dealing with conflicts, tradeoffs and priorities; evaluating the decisions impact centered on NFRs that commonly influence the success of ubiquitous applications; and systematically refining the models through the contributions specification for all alternatives on the NFRs. The NFR Framework offers graphs – **S**oftgoals **I**nterdependency **G**raphs (SIGs) – for NFRs modeling. SIGs represent NFRs as nodes; their refinements using AND/OR decompositions links; their positive and negative interdependencies as some+(*help*), some-(*hurt*), some++(*make*), some--(*break*) contribution links; their operationalizations as leaf nodes; and claims as annotations in natural language. Figure 4.6 illustrates a very simple SIG that models the Software Ubiquity, by considering its <u>decompositions</u> – AND links – in Software Pervasiveness, Software Mobility, and User Satisfaction; an <u>interdependency</u> between Software Mobility and User Satisfaction: Mobility[Software] positively impacts (*help*) on Satisfaction[User]; an <u>operationalization</u>: "Mobile Agents using special capabilities" *help* Mobility[Software]; and a <u>claim</u> "The software delegates the complex device's configuration activity to the user" *hurt* the decomposition between Ubiquity[Software] and Satisfaction[User]. It is also possible to analyze the SIG using propagation rules (e.g. if Pervasiveness[Software] AND Mobility[Software] AND Satisfaction[User] are *satisficed* (√), then Ubiquity[Software] is *satisficed*). It means "*satisfied in a certain degree*" (Yu 1997). The NFR Framework applied a qualitative approach to the evaluation of the NFRs represented in the SIG. Therefore, the impact of the decisions is qualitatively propagated through the graph by using propagation labels (i.e. from *denied* (χ) to *satisficed* ($\surd$)) to determine how well a chosen target system *satisfices* its NFRs.

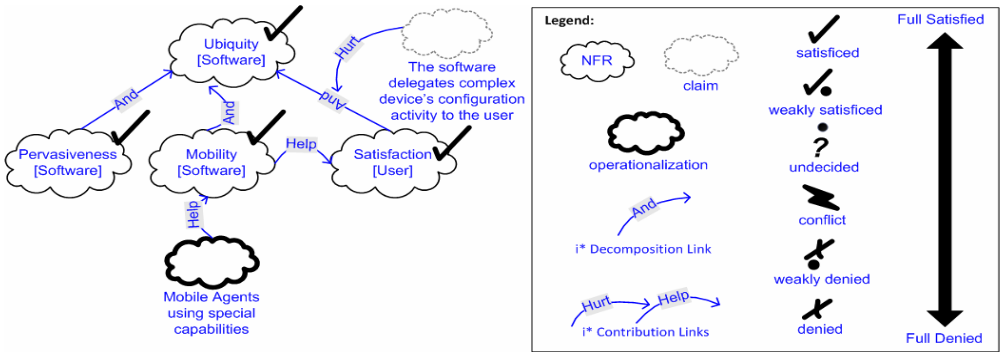


Figure 4.6 - Example of NFR SIG notation

The *NFR Catalogue Building Block* – Figure 4.7 – is composed of different *Ubiquity-Based Softgoals Interdependency Graphs* to both: (i) provide models reuse by extending and/or instantiating them from the proposed *SIGs Knowledge Base*; and (ii) guide the software engineers in the non-functional requirements elicitation, analysis and operationalization.

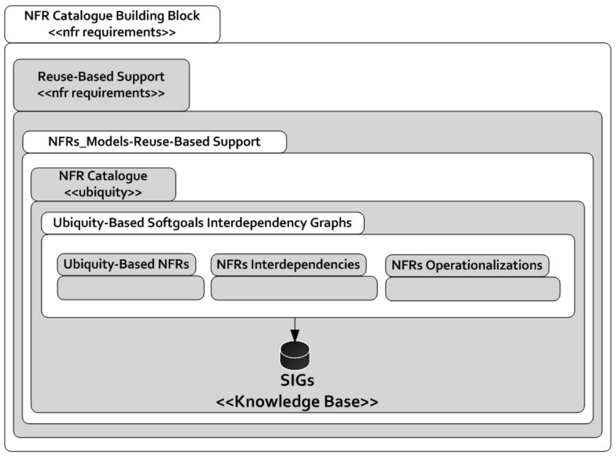

NFR Catalogue Building Block <<nfr requirements>>
Reuse-Based Support <<nfr requirements>>
NFRs_Models-Reuse-Based Support
NFR Catalogue <<ubiquity>>
Ubiquity-Based Softgoals Interdependency Graphs
Ubiquity-Based NFRs
NFRs Interdependencies
NFRs Operationalizations
SIGs <<Knowledge Base>>

Figure 4.7 - *NFR Catalogue Building Block* packages

In order to develop the catalogue, we concentrated our efforts on three activities: NFRs elicitation; NFRs decomposition; and NFRs interdependencies identification. These activities started from ubiquitous scenarios and the quality criteria identification obtained in the State-Of-The-Art, experts consultation, and during our experimental research. The elicited NFRs were evaluated with user participation. Those activities were iteratively performed, which allowed us to incrementally construct our knowledge base through the following phases:

(i) *State-Of-The-Art Investigation* - We started our work investigating the literature to compile an adequate initial understanding of ubiquitous concerns focusing on Ubiquitous Computing (e.g. (Weiser 1991; Weiser 1993; Abowd et al. 1998)) and experimentation-oriented papers (e.g. (Ravindran et al. 2002; Estrin et al. 2002; Scholtz and Consolvo 2004)). This investigation, conducted from different viewpoints (e.g. requirements and software engineers), allowed us to obtain a first version of the catalogue. It consisted of top-level ubiquitous requirements as well as their direct refinements. Specifically, the catalogue included three top-level NFRs (Ubiquity, Pervasiveness, and Mobility), and four refinement NFRs (Content Adaptability, Context Awareness, Device Heterogeneity, Software Processes Complexity Invisibility);

(ii) *Experimental Research* - Based on the initial version, we performed our first experimental research at the PUC-Rio Software Engineering Laboratory. Our main goal consisted of applying the first version of the catalogue's reusable models to the systematic development of ubiquitous applications. We obtained some interdependencies, and operationalizations for each specified NFR. Moreover, the research suggested some refinements for the first proposal, such as: we incorporated User Satisfaction as a seminal ubiquitous issue; and also other important NFRs as well as their refinements. Notable among them were Usability, Content/Service Accessibility, and Ubiquitous Profiles Awareness. As a result, the evolved catalogue constituted of 21 NFRs;

(iii) *Iterative Evolution* - During the last four years, from 2007 to 2010, we performed several iterations to evolve the catalogue. Basically, the catalogue's iterative evolution involved: (a) literature investigation; (b) catalogue content exploration; (c) catalogue content identification, considering ever-changing contexts, simulated by our case studies (e.g. some of them are presented in the beginning of this Chapter); (d) application of the catalogue on these case studies; (e) incremental refinement of the catalogue according to the newly discovered ubiquitous concerns; (f) comparative evaluation of refined and reusable models obtained from the catalogue to validate the refinements; and (g) catalogue evolution based on successful refinements. Our approach included developing other support sets to guide the design of ubiquitous applications, by dealing systematically with key ubiquitous issues. Basically, during this phase we iteratively created novel catalogue content, while eliminating replications, redundancies and ambiguous specifications. We also combined NFRs in one SIG to obtain a structured reusable model, while at other times we refined one NFR in different SIGs to improve our reusable models; and

(iv) *Evolution and Maintenance* – Throughout the process, collaborators could submit new SIGs and review the catalogue. This phase includes novel experimental research to incrementally refine the actual reusable models version, considering the use of our reusable models in different Ubiquitous Computing groups' projects. The catalogue's latest version consists of almost 700 interdependent softgoals (Table 4.1).

Table 4.1 - Summary of the main ubiquitous NFRs issues addressed by our models

NFR	Meaning	First Identification	Priority	Top Down View Category
Software Ubiquity	…	Phase 1	Extremely High	1
Software Pervasiveness	…	Phase 1	Extremely High	1
Software Mobility	…	Phase 1	Extremely High	1
Content Adaptability	…	Phase 1	Very High	2
Context Awareness	…	Phase 1	Very High	2
Device Heterogeneity	…	Phase 1	Very High	2
Process Complexity Invisibility	…	Phase 1	Very High	2
Software Distribution	…	Phase 1	Very High	2
User Satisfaction	…	Phase 2	Extremely High	1
Software Usability	…	Phase 2	High	3
Content/Service Accessibility	…	Phase 2	Very High	2
Ubiquitous Profiles Awareness	…	Phase 2	Very High	2
User Privacy	…	Phase 3 – First Iteration	High	3
Software Traceability	…	Phase 3 – First Iteration	Very High	2
Software Recoverability	…	Phase 3 – First Iteration	Very High	2
Software Portability	…	Phase 3 – First Iteration	Very High	2
Software Self-Regulation	…	Phase 3 – Second Iteration	High	3
Software Autonomy	…	Phase 3 – Second Iteration	High	3
Software Flexibility	…	Phase 3 – Second Iteration	High	3
Software Reactivity	…	Phase 3 – Second Iteration	High	3
...	...	...	...	...
Software Accuracy	…	Phase 3 – Last Iteration	High	3
Software Controllability	…	Phase 3 – Last Iteration	High	3
Software Transparency	…	Phase 3 – Last Iteration	Very High	2
New One	…	Phase 4	…	…

These softgoals are organized according to their importance for ubiquitous applications, obtained from our experimental research. The main softgoals – the most commonly found in the ubiquitous applications development process as well as the most generic ones – received highest priority. The catalogue is actually organized into four main softgoals (Ubiquity, Pervasiveness, Mobility, and User Satisfaction) at the top level. Moreover, there are 17 softgoals in the second level, including: Content Adaptability, Context Awareness, Device Heterogeneity, Transparency, and Process Complexity Invisibility. Furthermore, there are almost 200 NFRs at the third level, such as: Self-Regulation, Autonomy, Reactivity, and Controllability. This categorization – driven by the capturing of ubiquitous NFRs issues in several different ubiquitous applications – was applied to the entire catalogue, improving its applicability. It is important to notice that as the catalogue is in constant evolution, the refinements involve refactoring in the prioritizations and, consequently, they reflect on the catalogue's organization. Therefore, we are also proposing another way to organize our catalogue based on different criteria such as: most used NFRs or ones that address greater number of issues receive higher priorities.

Due to the huge number of NFRs and reusable models shared in our baseline, we also developed a Web-application to facilitate their access and to help in the presentation and browsing of its contents. This application offers different mechanisms to investigate and navigate - e.g. an exploration tree to navigate and choose the desired NFR, their meaning, and links to their SIGs and Frame-Like Notations. In addition, according to our experimental research, the NFRs' elicitation has been a good starting point for capitalizing knowledge in ubiquitous contexts, since they do not vary much from one ubiquitous application to another. It makes our reusable models as well as their decompositions, interdependencies, and operationalizations applicable to a broad class of ubiquitous applications. Moreover, we also developed a catalogue usage method, which is described in the detailed activity-based representation shown as follows. Figure 4.8 illustrates the meta-model of the used representation.

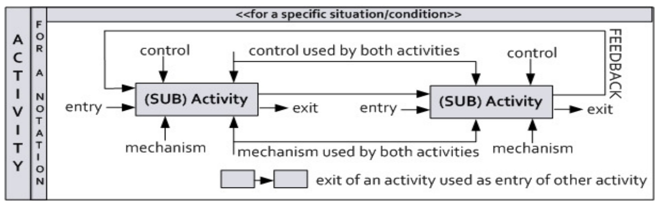

ACTIVITY
FOR A NOTATION
<<for a specific situation/condition>>
control
control used by both activities
control
FEEDBACK
entry
(SUB) Activity
exit
entry
(SUB) Activity
exit
mechanism
mechanism used by both activities
mechanism
exit of an activity used as entry of other activity

Figure 4.8 – Meta-model of the used activity-based representation

1. Explore activity (Figure 4.9): divided into Consult and Extract sub-activities. The Consult sub-activity consists of the catalogue knowledge investigation to understand ubiquitous concerns. The Extract sub-activity consists of the deduction of what knowledge is pertinent for the desired ubiquitous application.

The success of this sub-activity depends on whether the Consult is satisfactorily accomplished.

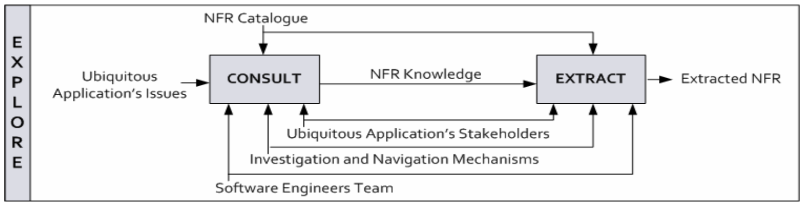

EXPLORE
NFR Catalogue
Ubiquitous Application's Issues
CONSULT
NFR Knowledge
EXTRACT
Extracted NFR
Ubiquitous Application's Stakeholders
Investigation and Navigation Mechanisms
Software Engineers Team

Figure 4.9 - *NFR Catalogue Usage Method – Explore* activity

The catalogue mainly helped us with regard to knowledge capitalization – by providing resources to search it and to navigate through its interdependencies – and terms familiarization. Contributing to this field, the catalogue presents the meaning of all baseline terms and information sources for further and deeper investigations. After this exploratory searching, we were able to identify the main NFR-related issues; determine their impacts on the concern under analysis; and capitalize sufficient knowledge to put together a comprehensive view of ubiquitous main concerns.

2. Collect activity (Figure 4.10): composed of Pick-up and Instantiate/Evolve sub-activities. The Pick-up activity occurs if the extracted knowledge matches with the ubiquitous application's needs. Thus, it is not necessary to perform the next activity – i.e. Model activity – and the software engineers can directly go to the Operationalize activity. If adjustments are necessary, the Instantiate/Evolve sub-activity is performed. Thus, the knowledge in SIG and Frame-Like Notation is instantiated and evolved.

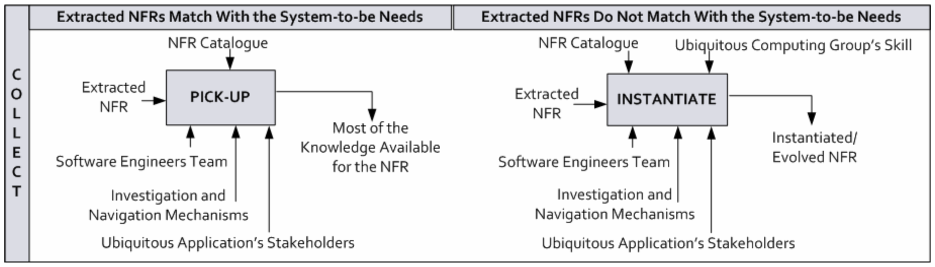

COLLECT
Extracted NFRs Match With the System-to-be Needs
Extracted NFRs Do Not Match With the System-to-be Needs
NFR Catalogue
Extracted NFR
PICK-UP
Most of the Knowledge Available for the NFR
Software Engineers Team
Investigation and Navigation Mechanisms
Ubiquitous Application's Stakeholders
NFR Catalogue
Ubiquitous Computing Group's Skill
Extracted NFR
INSTANTIATE
Instantiated/ Evolved NFR
Software Engineers Team
Investigation and Navigation Mechanisms
Ubiquitous Application's Stakeholders

Figure 4.10 - *NFR Catalogue Usage Method – Collect* activity

3. Model activity (Figure 4.11): based on Decompose and Determine Interdependencies in SIG Notation; and Specify Decomposition, Claim, Correlation Rule in Frame-Like Notation – depending on the chosen notation. It is also possible to use both. The first is a graphical view whereas the second is a

semi-structured specification centered on parent, offspring, contribution, constraint and condition. Both SIG and Frame-Like notations are based on the NFR Framework conceptual model (Chung et al. 2000).

SIG Notation: In the Decompose sub-activity, it is possible to decompose NFRs that were instantiated or evolved in the Collect activity. In the Determine Interdependencies sub-activity, it is necessary to determine the interdependencies among the instantiated/evolved NFRs as well as their decomposed NFRs by using contribution links (e.g. some+ (*help*), some- (*hurt*), some++ (*make*) and some-- (*break*)).

FRAME-LIKE Notation: In the Specify Decomposition sub-activity, it is possible to specify the parent, offspring (e.g. decomposed NFRs) and contribution (e.g. *help* and *hurt*) for the NFRs that were instantiated/evolved in the Collect activity. In the Specify Claim sub-activity, the specification is focused on parent (e.g. nfr_1 AND nfr_2 AND nfr_3 SATISFICE nfr_{parent}), offspring (e.g. Claim[argument]), contribution and constraint (e.g. /*argument*/). Finally, in the Specify Correlation Rule sub-activity, the specification is based on parent, offspring, contribution and condition (e.g. true and false) between the instantiated/evolved NFRs and their decomposed NFRs.

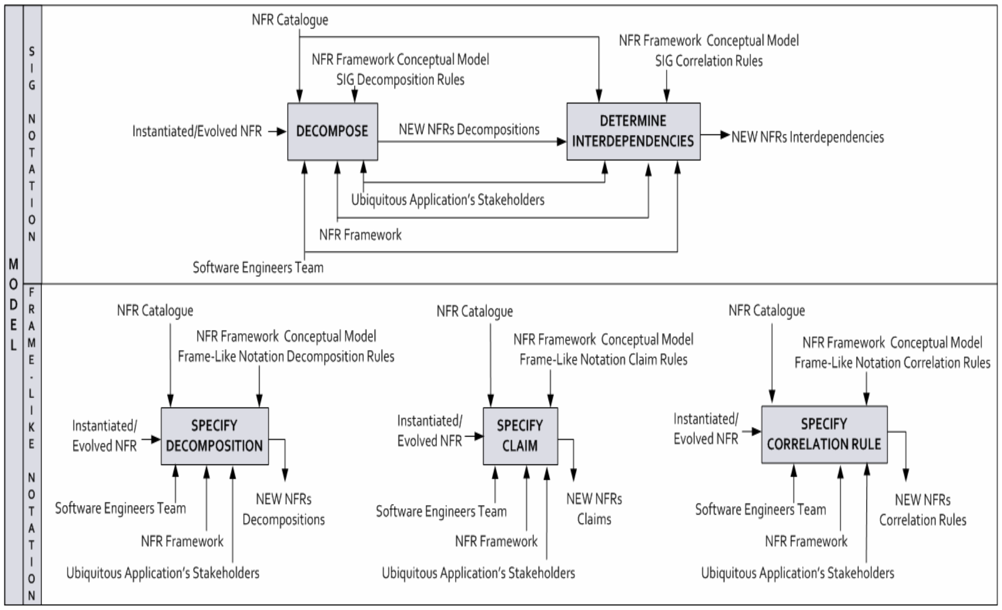

MODEL
SIG NOTATION
NFR Catalogue
NFR Framework Conceptual Model
SIG Decomposition Rules
NFR Framework Conceptual Model
SIG Correlation Rules
Instantiated/Evolved NFR
DECOMPOSE
NEW NFRs Decompositions
DETERMINE INTERDEPENDENCIES
NEW NFRs Interdependencies
Ubiquitous Application's Stakeholders
NFR Framework
Software Engineers Team
FRAME-LIKE NOTATION
NFR Catalogue
NFR Framework Conceptual Model
Frame-Like Notation Decomposition Rules
Instantiated/ Evolved NFR
SPECIFY DECOMPOSITION
NEW NFRs Decompositions
Software Engineers Team
NFR Framework
Ubiquitous Application's Stakeholders
NFR Catalogue
NFR Framework Conceptual Model
Frame-Like Notation Claim Rules
Instantiated/ Evolved NFR
SPECIFY CLAIM
NEW NFRs Claims
Software Engineers Team
NFR Framework
Ubiquitous Application's Stakeholders
NFR Catalogue
NFR Framework Conceptual Model
Frame-Like Notation Correlation Rules
Instantiated/ Evolved NFR
SPECIFY CORRELATION RULE
NEW NFRs Correlation Rules
Software Engineers Team
NFR Framework
Ubiquitous Application's Stakeholders

Figure 4.11 - *NFR Catalogue Usage Method – Model* activity

4. Operationalize Activity (Figure 4.12): to define an adequate set of operationalizations for further implementations. Our catalogue already offers

some operationalizations to be reused and attend needs faster. In order to operationalize with the NFR Catalogue support by simply selected and picked up operationalizations from the baseline, the method suggests to perform the Select Operationalizations Based On NFR Catalogue sub-activity. However, it is also possible to establish new support by using developer expertise. Therefore, the method suggests the Specify Operationalizations Not Based On NFR Catalogue sub-activity.

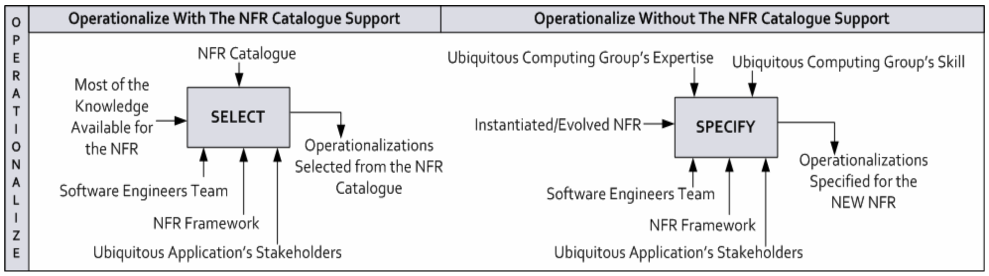

OPERATIONALIZE
Operationalize With The NFR Catalogue Support
Operationalize Without The NFR Catalogue Support
NFR Catalogue
Most of the Knowledge Available for the NFR
SELECT
Operationalizations Selected from the NFR Catalogue
Software Engineers Team
NFR Framework
Ubiquitous Application's Stakeholders
Ubiquitous Computing Group's Expertise
Ubiquitous Computing Group's Skill
Instantiated/Evolved NFR
SPECIFY
Operationalizations Specified for the NEW NFR
Software Engineers Team
NFR Framework
Ubiquitous Application's Stakeholders

Figure 4.12 - *NFR Catalogue Usage Method – Operationalize* activity

5. Validate activity (Figure 4.13): divided into Evaluate and Solve Conflicts sub-activities. The Evaluate sub-activity mainly checks interdependencies using correlation rules and stakeholders' meetings by identifying possible conflicts (e.g. to satisfy the parent NFR – e.g. nfr_{parent} – the decomposed NFRs must be *satisficed* (√) – i.e. *Satisficed* means "*satisfied in a certain degree*" (Yu 1997)). The Solve-Conflicts sub-activity deals with conflicts and open states by solving them with alternative interdependencies/operationalizations.

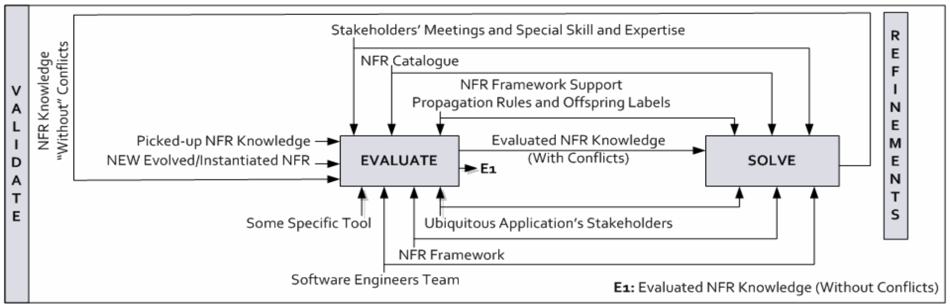

VALIDATE
NFR Knowledge "Without" Conflicts
Stakeholders' Meetings and Special Skill and Expertise
NFR Catalogue
NFR Framework Support
Propagation Rules and Offspring Labels
Picked-up NFR Knowledge
NEW Evolved/Instantiated NFR
EVALUATE
E1
Evaluated NFR Knowledge (With Conflicts)
SOLVE
REFINEMENTS
Some Specific Tool
Ubiquitous Application's Stakeholders
NFR Framework
Software Engineers Team
E1: Evaluated NFR Knowledge (Without Conflicts)

Figure 4.13 - *NFR Catalogue Usage Method – Validate* activity

Furthermore, the method contemplates the feedback notion, allowing refinements when misconception/misunderstanding occurs from faulty judgment, deficient knowledge or lack of forethought. Thus, it is necessary to constantly return to previous activities to review details.

4.3. Integration Building Block

Another reuse-based mechanism proposed in our approach is the *Integration Building Block* – to integrate heterogeneous devices with the MAS platform – centered on the JADE-LEAP platform (**J**ava **A**gent **D**evelopment **E**nvironment-**L**ightweight **E**xtensible **A**gent **P**latform) (Caire 2003). It is an extension for the JADE platform to deal with heterogeneous mobile devices (e.g. simple cell-phones and Smartphones). It allows the development of FIPA-compliant MASs in these devices, which are normally limited in terms of memory and processing capacities. Therefore, it provides execution modes to integrate heterogeneous devices (e.g. jse-based devices[2], PJava devices[3] and MIDP devices[4]) with the MAS platform. In this work our attention is on two specific execution modes, the standalone and the split modes. The standalone mode integrates the PJava devices with the MAS platform. These devices are powerful in memory and processing capacities. Thus, they are able of running the platform's container by using their own resources. In this case, a complete container is executed on the device. The split mode integrates the MIDP devices – i.e. the great majority of Java enabled cell-phones – with the MAS platform. In this mode, the container is split into a *Front-End* and a *Back-End*. Both, *Front-End* and *Back-End*, are linked through the wireless connection. Moreover, the *Front-End* runs on the MIDP device and the *Back-End* runs on a powerful machine (normally a jse host). As the *Front-End* is lighter than a complete container, the split execution mode is interesting for limited mobile devices, which are constrained in memory and processing capacities. In both modes the device's user is not concerned about the integration process.

The integration process is performed by the agents without disturbing the user or even distracting her/him (Weiser and Brown 1995), which contributes to the invisibility – a quality criterion important for the user satisfaction and a social implication of Ubiquitous Computing (Langheinrich 2001). Moreover, the JADE-LEAP agent is registered and its life-cycle is controlled by specific services of the

[2] Desktops and Notebooks with jdk1.2 or superior.
[3] PDAs that run **P**ersonal **J**ava.
[4] Mobile phones that support the MIDP (**M**obile **I**nformation **D**evice **P**rofile) – a Java runtime environment for mobile devices.

platform, respectively the **D**irectory **F**acilitator (DF) Service – i.e. *Yellow Pages* – and the **A**gent **M**anagement **S**ystem (AMS) Service – i.e. *White Pages*. These services improve the dependability, accountability and security of the agents' activities by also contributing to the safety of the user data management.

Our *Integration Building Block* based on the JADE-LEAP platform is composed of the *Reuse-Based Support* package (Figure 4.14), in which the main sub-package is the "Invisibility-Based Support." This sub-package supports the ubiquitous application invisibility by providing, for example, the "Platform Integration Support." This latter support consists of an API to deal with the integration of different devices with the intentional MAS platform. A JADE-LEAP Agent, which is a behavioral agent, performs the integration process. In this case, we used a behavior-based agent instead of an intentional agent, as the former agent is lighter than the latter one. It avoids problems with the limited nature of mobile devices with low memory and processing capacities. Moreover, the integration demands the support offered in both sub-packages "Split Execution Mode" and "Standalone Execution Mode" to respectively integrate MIDP and PJava devices with the intentional MAS platform. Furthermore, the usage of a JADE-LEAP agent improves the accountability in case of future investigation based on some unapproved/unattested conduction during the integration process as: (i) this agent is registered into the MAS platform with a unique identifier centered on the *DF Service*; (ii) there is one agent responsible for each integration process; and (iii) the agent's life-cycle is totally controlled by the services offered into the platform. It allows, among other contributions, to monitor the agent "on the fly," by also tracing its activities.

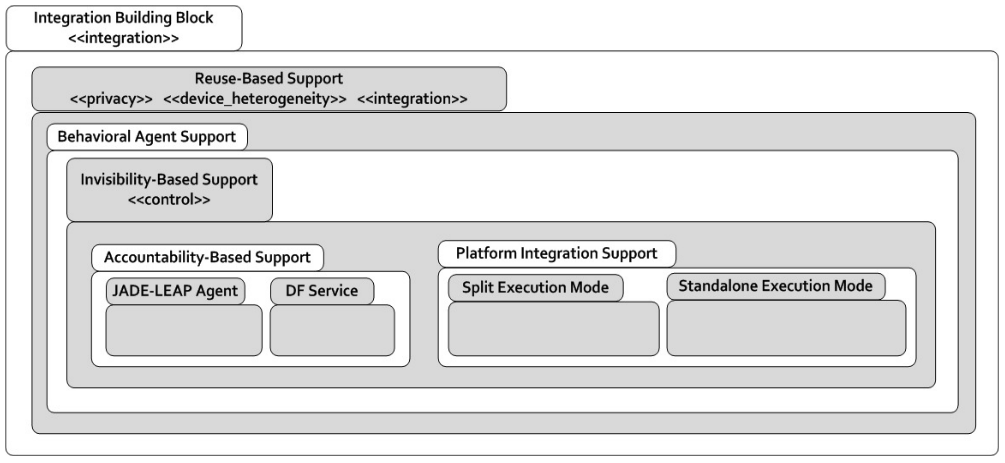

Integration Building Block
<<integration>>
Reuse-Based Support
<<privacy>> <<device_heterogeneity>> <<integration>>
Behavioral Agent Support
Invisibility-Based Support
<<control>>
Accountability-Based Support
JADE-LEAP Agent
DF Service
Platform Integration Support
Split Execution Mode
Standalone Execution Mode

Figure 4.14 - *Integration Building Block* packages

4.4. Intentional Agents' Reasoning Building Block

In order to develop intentional agents centered on a BDI-based reasoning engine, we focused our attention on the JADEX Framework (Braubach et al. 2003; Braubach et al. 2004). This framework is developed by a research group at Hamburg University. It is an API that provides support to develop MAS applications centered on the BDI model. This model is based on the belief, desire and intention abstractions. The belief represents the agent's knowledge – i.e. its informational state based on its beliefs about the world. The desire is the agent's goal – i.e. its motivational state. Finally, the intention is an action or a sequence of actions to achieve this goal – i.e. its deliberative state. Therefore, the BDI-based agents are intentional and capable of acting in a goal-oriented manner by using their beliefs, desires and intentions. The JADEX reasoning engine provides useful methods – by extending abstract classes as plan Java classes – for dispatching goals, sub-goals and events; sending messages and awaiting internal events. During this process, the agent's beliefs base can be modified – by manipulating the stored facts – to update the agent's knowledge by respecting the context at runtime. In order to improve the context-aware agents, the JADEX uses an XML-based **A**gent **D**efinition **F**ile (ADF), which specifies the initial beliefs, desires and intentions. The engine uses this file – at runtime – to instantiate an agent model.

The mentioned infrastructure can be pertinent in the development of privacy-based ubiquitous applications. For example, a correct management of the user profiles – as previously mentioned – is a complex task when we consider highly everywhere/anywhere-applications. The need for high performance combined with the necessity of guaranteeing security, integrity and dependability for sensitive profile information often results in a trade-off between distributed and centralized approaches. This trade-off is even more critical if it is necessary to update ubiquitous profiles in a dynamic way. In these scenarios, intentional agents can be used to make decisions on whether to present the service or not by profile-matching between different profiles and the users' preferences or the service providers' business rules. It intends to provide personalized service. In addition, if the profiles evolve over time, intentional agents can also evolve their beliefs base by, for example, representing the users and/or the service providers as personal assistants, conscious about the context. Moreover, those agents have specific and useful properties, such as: autonomy, reactivity, proactivity, mobility, reasoning capacity, learning capacity and adaptability. Therefore, the development of these entities based on the BDI model may confer to them the ability of preserving the confidentiality of the user profile information and the service provider's commercial strategies.

In addition, the agent's functionalities compose reusable modules – called *Capabilities* in the JADEX. These modules can be plugged into existing agents by improving their capacity in the MAS platform, even at runtime. Examples of specific functionalities are the capabilities to allow: the agent's dynamic creation and mobility and the agent's dynamic learning based on the services offered by the MAS platform. The agent's creation and mobility performed at runtime contribute to the invisibility of the application by improving the agent's autonomy condition to act and to respond the context situation without the human intervention, which in our approach means: "without disturbing people." Moreover, the mobility directly deals with the location and proximity concern – i.e. according to (Langheinrich 2001), it is a principle of privacy. Finally, the possibility of learning how to use a service and how to interact with the agents of this service by respecting specific business rules of the service provider at runtime can be viewed as an interesting mechanism for ever-changing environments.

Concentrating our attention on the JADEX-based resources, we propose the *Intentional Agents' Reasoning Building Block*. This support set is composed of the *Reuse-Based Support* package (Figure 4.15), in which the main sub-package is the "Invisibility-Based Support." Different resources confer on this package the ability to balance the invisibility and transparency and the personalization and privacy issues. In this scenario, we can mention the "Autonomy- Reactivity- Proactivity- Mobility- & Adaptability-Based Support" package and its main sub-package: "Intentional Multi-Agent Systems Support." This sub-package consists of an API to support the development of *JADEX Agents* centered on the *BDI Model*, the *FIPA Standards Ontological Support* (Bellifemine et al. 2007; Serrano and Lucena 2010b) and specific *JADEX Capabilities*.

The *BDI Model* is also part of the *Agents' Cognitive-Ability-Based Support*, which offers resources to improve the cognitive ability of the *JADEX Agents* centered on the intentionality abstraction. The *FIPA Standards*

Ontological Support is part of the *Agents' Interoperability-Based Support*, whose resources (Serrano and Lucena 2010b) contribute to the agents' communication and inter-operability by using ontologies. Moreover, the success of the agents' communication demands ubiquitous profiles investigation by including the users' privacy preferences "capturing" at runtime. The *JADEX Capabilities* is part of the *Dependability- Accountability- & Security-Based Support*, which provides *Specific Capability Support*, such as: (i) to improve the accountability by using the *DF Capability* to register and deregister the platform's agents with a unique identifier that may be used to determine which agent is dealing or dealt with the user data. The identification of a specific agent can be dynamically performed anywhere and at any time, which confers the user the possibility to interact with the application by trusting it; and (ii) to allow that an intentional agent moves from one container to another by using the *Mobility Capability* in order to perform complex services in a dedicated server. It also improves the invisibility issue.

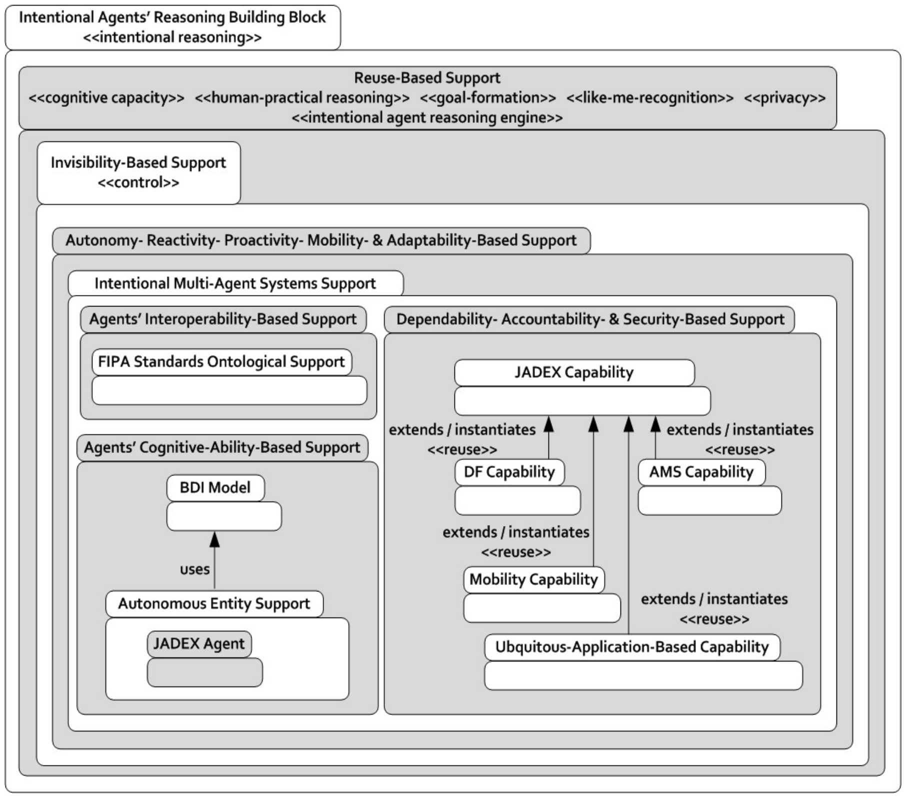

Intentional Agents' Reasoning Building Block
<<intentional reasoning>>
Reuse-Based Support
<<cognitive capacity>> <<human-practical reasoning>> <<goal-formation>> <<like-me-recognition>> <<privacy>>
<<intentional agent reasoning engine>>
Invisibility-Based Support
<<control>>
Autonomy- Reactivity- Proactivity- Mobility- & Adaptability-Based Support
Intentional Multi-Agent Systems Support
Agents' Interoperability-Based Support
FIPA Standards Ontological Support
Agents' Cognitive-Ability-Based Support
BDI Model
uses
Autonomous Entity Support
JADEX Agent
Dependability- Accountability- & Security-Based Support
JADEX Capability
extends / instantiates
<<reuse>>
DF Capability
AMS Capability
extends / instantiates
<<reuse>>
Mobility Capability
extends / instantiates
<<reuse>>
Ubiquitous-Application-Based Capability
extends / instantiates
<<reuse>>

Figure 4.15 - *Intentional Agents' Reasoning Building Block* packages

Furthermore, we also improve the agents' reasoning to specifically deal with non-functional requirements (e.g. privacy and their correlated issues, such as: dependability, accountability, integrity and security) at runtime by instantiating –

based on the cognitive domain and the application under analysis – the Fuzzy-Logic Library proposed in (Bigus and Bigus 2001). This library provides resources to the specification of fuzzy-logic conditional rules. Based on this library, we constructed the *Fuzzy-Logic-Based Support* package (Figure 4.16) to complement the *Intentional Agents' Reasoning Building Block*. This package provides resources for the proposed approach to deal with quality criteria at runtime. Simplifying the process, intentional agents basically analyze their belief base, the ubiquitous profiles and additionally run those fuzzy-logic conditional rules – "on the fly" – to choose an alternative task that better satisfy a specific user by considering the specified non-functional requirements of the ubiquitous application under analysis and the user preferences. All the process is performed at runtime without disturbing the user – i.e. the user may even be unaware that it is actually being performed. Only to illustrate, consider two specific criteria: security and price. Following the described process is possible to determine – at runtime – that the security, for example, is more relevant than the price by considering the user under analysis. Therefore, the agent can choose an alternative task that minimizes the impact on security.

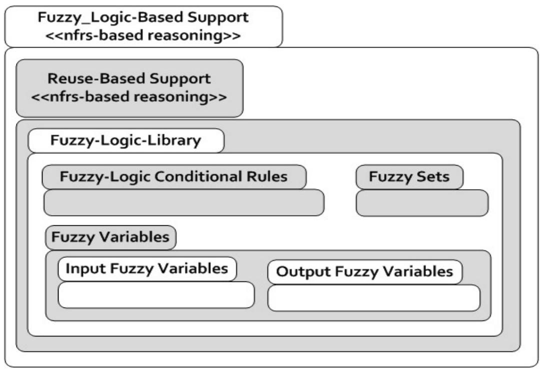


Figure 4.16 - *Fuzzy-Logic-Based Support* complementary package

4.5. Dynamic Interface Construction Building Block

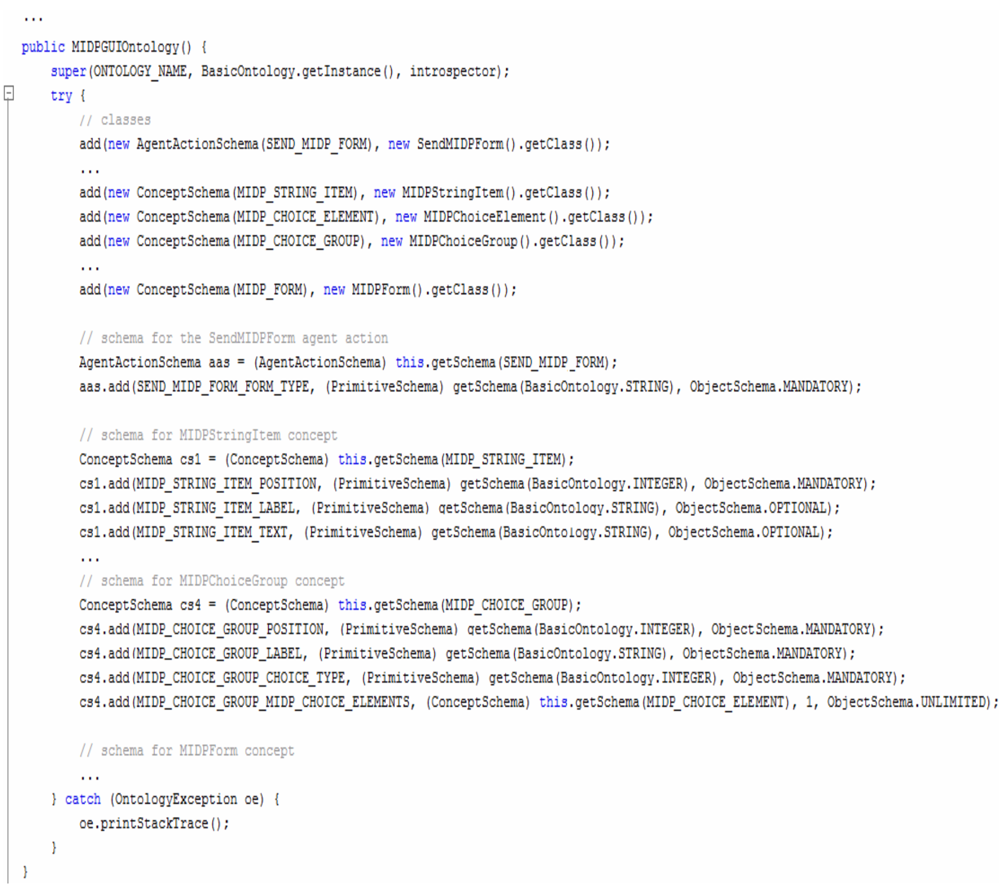
We also offer an ontology-based mechanism to improve the agents' communication and inter-operability and the dynamic interface construction. We published a detailed view of how to apply FIPA Standards Ontological Support to intentional-MAS-oriented ubiquitous applications in (Serrano and Lucena 2010b; Serrano and Lucena 2011b). According to the FIPA SL Codec (Bellifemine et al. 2007), the ontology is composed of the vocabulary and the nomenclature. The vocabulary describes the concepts terminology. These concepts are used by the agents in the interaction among them. The nomenclature describes the concepts semantic and structure, and depends on the relationships among these concepts. In order to implement the ontology, we had to extend the classes *BasicOntology* and *ACLOntology*, predefined in the FIPA SL Codec, by adding the elements schemas that describe the structure of the concepts, agent actions, and predicates of the exchanged messages. The *Concept*, *AgentAction*, and *Predicate* are interfaces, which correlated classes are *ConceptSchema*, *AgentActionSchema*, and *PredicateSchema*. In fact, these interfaces have a super-class called *ObjectSchema*. As follows, we have a brief description of *Concept*, *AgentAction*, and *Predicate*:

- *Concept* represents expressions that indicate entities with a complex structure, such as: (User :id 000000 :name James :address "1111 Something Avenue"). It means that there is a user with the id 000000, the name James, and the address 1111 Something Avenue;
- *AgentAction* represents concepts that indicate actions performed by the agents in the MAS platform, such as: (Request (Registration :Web site "Music Store") (User :id 000000)). It means that the user with the id 000000 requests the registration for the web site "Music Store"; and
- *Predicate* represents expressions that inform some detail about the status of the world, such as: (Is-user-of (User :id 000000) (Web site :name "Music Store")). It means that the user with the id 000000 is user of the Web site, which name is "Music Store".

We are particularly following the reference model proposed by Fabio Bellifemine, Giovanni Caire, and Dominic Greenwood in (Bellifemine et al. 2007). In different ubiquitous applications, we have, for example, *Elements* in the interface level; cognitive domain level, and application level. We firstly defined an ontological Java class that extends the *Ontology* class for each interface *Element* in the application's context. Each ontological Java class is declared as a singleton object as this class is normally not evolved during the agent's lifetime. For the same reason, we defined another Java class, which also extends the *Ontology* class, and contains a static method in order to access this singleton object. It means that different software agents that are in the same Java Virtual

Machine can share the same ontology object. An example of ontological Java classes in the interface level is presented as a code fragment in Figure 4.17.

Each element in a schema has a name and a type. An element can be declared as "OPTIONAL" or "MANDATORY." An "OPTIONAL" element means it can assume a "null" value. On the other hand, a "MANDATORY" element means that an *OntologyException* will be thrown if a "null" value was found. An element in a schema can also be a list, in which, for example, the cardinality of this element is zero or more *String* type elements.



```
...
public MIDPGUIOntology() {
    super(ONTOLOGY_NAME, BasicOntology.getInstance(), introspector);
    try {
        // classes
        add(new AgentActionSchema(SEND_MIDP_FORM), new SendMIDPForm().getClass());
        ...
        add(new ConceptSchema(MIDP_STRING_ITEM), new MIDPStringItem().getClass());
        add(new ConceptSchema(MIDP_CHOICE_ELEMENT), new MIDPChoiceElement().getClass());
        add(new ConceptSchema(MIDP_CHOICE_GROUP), new MIDPChoiceGroup().getClass());
        ...
        add(new ConceptSchema(MIDP_FORM), new MIDPForm().getClass());

        // schema for the SendMIDPForm agent action
        AgentActionSchema aas = (AgentActionSchema) this.getSchema(SEND_MIDP_FORM);
        aas.add(SEND_MIDP_FORM_FORM_TYPE, (PrimitiveSchema) getSchema(BasicOntology.STRING), ObjectSchema.MANDATORY);

        // schema for MIDPStringItem concept
        ConceptSchema cs1 = (ConceptSchema) this.getSchema(MIDP_STRING_ITEM);
        cs1.add(MIDP_STRING_ITEM_POSITION, (PrimitiveSchema) getSchema(BasicOntology.INTEGER), ObjectSchema.MANDATORY);
        cs1.add(MIDP_STRING_ITEM_LABEL, (PrimitiveSchema) getSchema(BasicOntology.STRING), ObjectSchema.OPTIONAL);
        cs1.add(MIDP_STRING_ITEM_TEXT, (PrimitiveSchema) getSchema(BasicOntology.STRING), ObjectSchema.OPTIONAL);
        ...
        // schema for MIDPChoiceGroup concept
        ConceptSchema cs4 = (ConceptSchema) this.getSchema(MIDP_CHOICE_GROUP);
        cs4.add(MIDP_CHOICE_GROUP_POSITION, (PrimitiveSchema) getSchema(BasicOntology.INTEGER), ObjectSchema.MANDATORY);
        cs4.add(MIDP_CHOICE_GROUP_LABEL, (PrimitiveSchema) getSchema(BasicOntology.STRING), ObjectSchema.MANDATORY);
        cs4.add(MIDP_CHOICE_GROUP_CHOICE_TYPE, (PrimitiveSchema) getSchema(BasicOntology.INTEGER), ObjectSchema.MANDATORY);
        cs4.add(MIDP_CHOICE_GROUP_MIDP_CHOICE_ELEMENTS, (ConceptSchema) this.getSchema(MIDP_CHOICE_ELEMENT), 1, ObjectSchema.UNLIMITED);

        // schema for MIDPForm concept
        ...
    } catch (OntologyException oe) {
        oe.printStackTrace();
    }
}
```

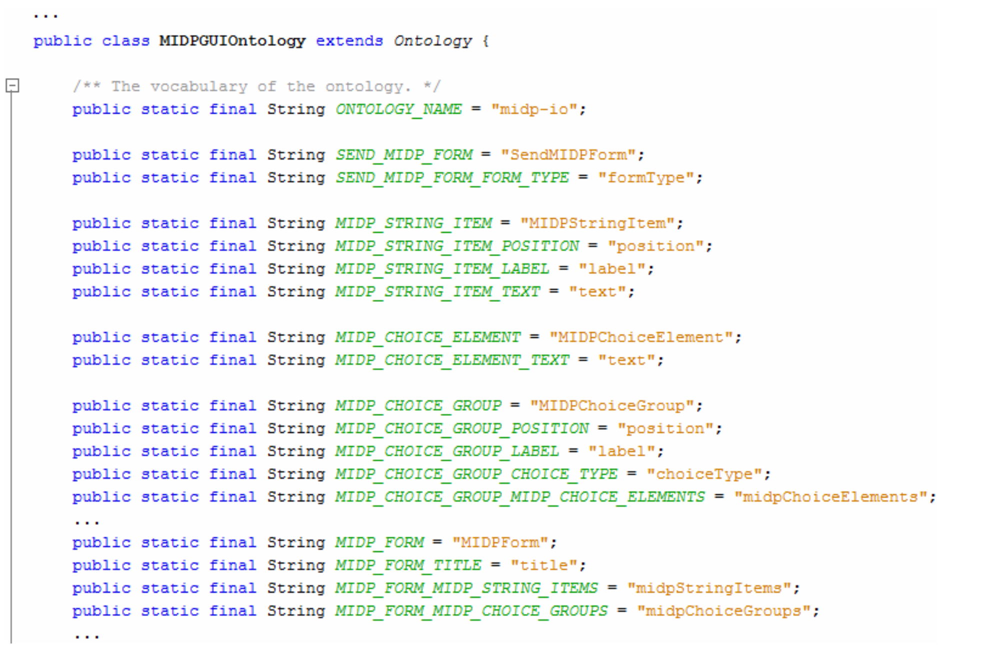
Figure 4.17 - Ontological Java class

Only to clarify the idea, some interface *Elements* – used by an Interface Agent to dynamically construct forms that will be presented to the user using her/his own device and according to the her/his preferences and the devices features (e.g. memory/processing capacities, screen size, and resolution) – are:

- *SendMIDPForm* agent action: is the ontological representation for an action performed by the Interface Agent in order to send a form;
- *MIDPStringItem* concept: is an ontological concept that describes a *StringItem* element, which can be used to compose the *Form*, by representing a *spring*;

- *MIDPChoiceElement* concept: is an ontological concept that describes a *ChoiceElement*, which can be used to compose the *ChoiceGroup*, by representing the alternative text;
- *MIDPChoiceGroup* concept: is an ontological concept that describes a *ChoiceGroup*, which can be used to compose the *Form*, by representing a group of choices. Moreover, it can be composed of one or more *ChoiceElement(s)*;
- *MIDPImage* concept: is an ontological concept that represents an *Image*, which can be adapted based on the device features to compose the *Form*; and
- *MIDPForm* concept: is an ontological concept that describes a *Form*, which can be composed of zero or more *StringItem(s)*, *ChoiceGroup(s) and Image(s).*

The ontological Java class implements the Vocabulary Java class, which code fragment is illustrated in Figure 4.18.



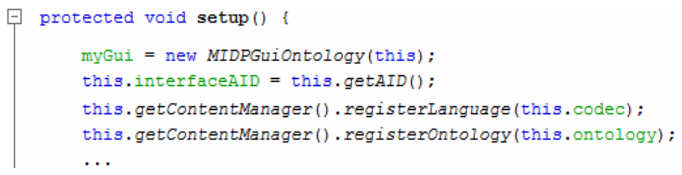
```
...
public class MIDPGUIOntology extends Ontology {

    /** The vocabulary of the ontology. */
    public static final String ONTOLOGY_NAME = "midp-io";

    public static final String SEND_MIDP_FORM = "SendMIDPForm";
    public static final String SEND_MIDP_FORM_FORM_TYPE = "formType";

    public static final String MIDP_STRING_ITEM = "MIDPStringItem";
    public static final String MIDP_STRING_ITEM_POSITION = "position";
    public static final String MIDP_STRING_ITEM_LABEL = "label";
    public static final String MIDP_STRING_ITEM_TEXT = "text";

    public static final String MIDP_CHOICE_ELEMENT = "MIDPChoiceElement";
    public static final String MIDP_CHOICE_ELEMENT_TEXT = "text";

    public static final String MIDP_CHOICE_GROUP = "MIDPChoiceGroup";
    public static final String MIDP_CHOICE_GROUP_POSITION = "position";
    public static final String MIDP_CHOICE_GROUP_LABEL = "label";
    public static final String MIDP_CHOICE_GROUP_CHOICE_TYPE = "choiceType";
    public static final String MIDP_CHOICE_GROUP_MIDP_CHOICE_ELEMENTS = "midpChoiceElements";
    ...
    public static final String MIDP_FORM = "MIDPForm";
    public static final String MIDP_FORM_TITLE = "title";
    public static final String MIDP_FORM_MIDP_STRING_ITEMS = "midpStringItems";
    public static final String MIDP_FORM_MIDP_CHOICE_GROUPS = "midpChoiceGroups";
    ...
```

Figure 4.18 - Ontology vocabulary for *Interface Elements*

As presented on (Bellifemine et al. 2007), the next three steps are necessary to conclude the ontology: *(i)* define the content language; (ii) register the content language and the ontology using a software agent; and *(iii)* create or manipulate the content expressions as Java Objects.

i. The first step consists of defining the content language. Using the FIPA Coder and Decoder we have the possibility to choose the SL Language or the LEAP language. It is also possible to develop an agent that uses a proper language by implementing the *jade.content.lang.Codec* interface. The SL Language is a human-readable content language, which content expression is a *string*. The LEAP language is a non-human-readable content language, which content expression is a sequence of bytes. Moreover, the LEAP language is lighter than the SL language. This feature is particularly interesting in strong memory and processing limitations. In order to illustrate our proposal, we used the SL language.

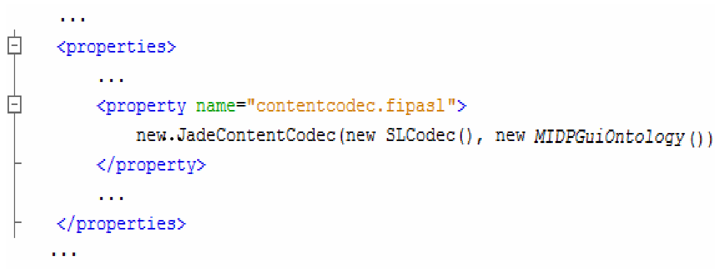
ii. The second step consists of registering the content language and the ontology using a software agent. Normally, in behavior-based agents, this registration is performed in the agent *setup()* method as presented in Figure 4.19 for the *Interface Agent* – a JAVA code fragment. As this Interface Agent runs inside the MIDP device, we decided to use a "light" agent, based on behavior to avoid problems with the device memory and processing limitations.



```
protected void setup() {
    myGui = new MIDPGuiOntology(this);
    this.interfaceAID = this.getAID();
    this.getContentManager().registerLanguage(this.codec);
    this.getContentManager().registerOntology(this.ontology);
    ...
```

Figure 4.19 - Registering content language and ontology using a behavioral *Interface Agent*

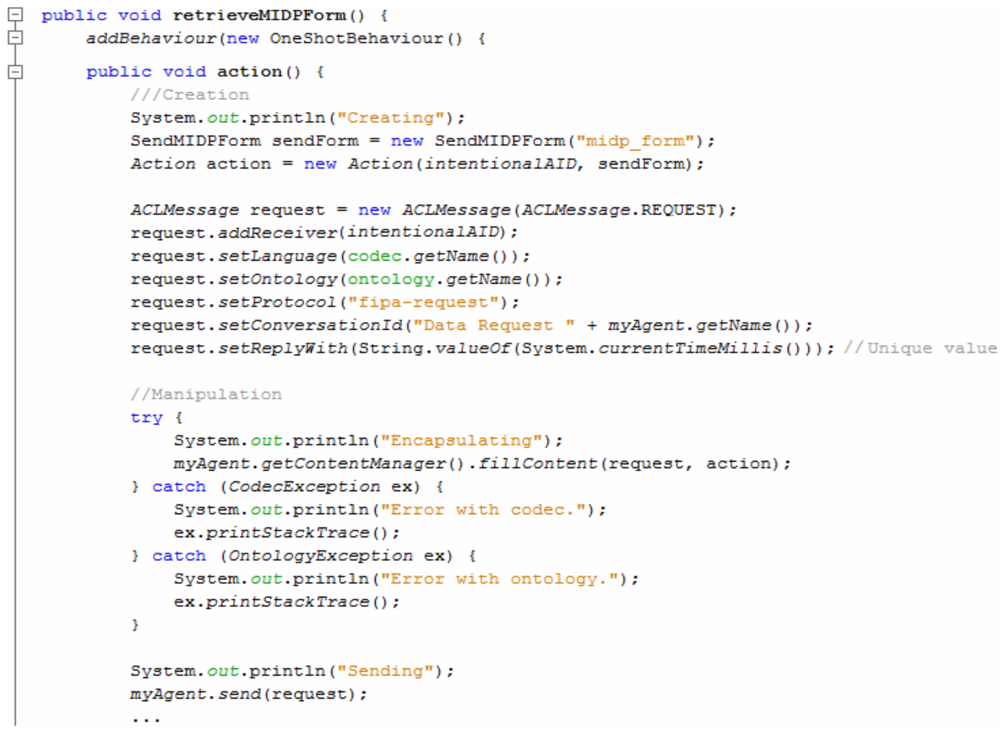
However, as we are focusing on using intentional agents to improve the cognitive capacity, the "like me" recognition, and the goal formation, we also registered the content language and the ontology according to the JADEX specifications and the BDI notation as shown in Figure 4.20 – XML code fragment of the *Intentional Agent* (*property tag*).



```
 ...
<properties>
    ...
    <property name="contentcodec.fipasl">
        new.JadeContentCodec(new SLCodec(), new MIDPGuiOntology())
    </property>
    ...
</properties>
...
```

Figure 4.20 - Registering content language and ontology using an *Intentional Agent*

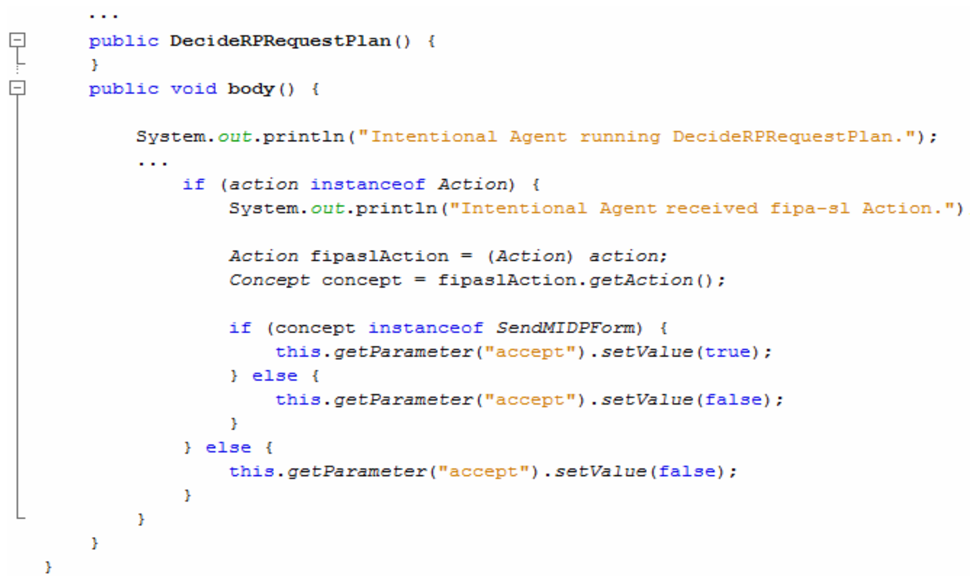
iii. The third step consists in creating and manipulating the content expressions as Java Objects. Figure 4.21 shows the code fragment about this step using the *Interface Agent*.



```
public void retrieveMIDPForm() {
    addBehaviour(new OneShotBehaviour() {

    public void action() {
        ///Creation
        System.out.println("Creating");
        SendMIDPForm sendForm = new SendMIDPForm("midp_form");
        Action action = new Action(intentionalAID, sendForm);

        ACLMessage request = new ACLMessage(ACLMessage.REQUEST);
        request.addReceiver(intentionalAID);
        request.setLanguage(codec.getName());
        request.setOntology(ontology.getName());
        request.setProtocol("fipa-request");
        request.setConversationId("Data Request " + myAgent.getName());
        request.setReplyWith(String.valueOf(System.currentTimeMillis())); //Unique value

        //Manipulation
        try {
            System.out.println("Encapsulating");
            myAgent.getContentManager().fillContent(request, action);
        } catch (CodecException ex) {
            System.out.println("Error with codec.");
            ex.printStackTrace();
        } catch (OntologyException ex) {
            System.out.println("Error with ontology.");
            ex.printStackTrace();
        }

        System.out.println("Sending");
        myAgent.send(request);
        ...
```

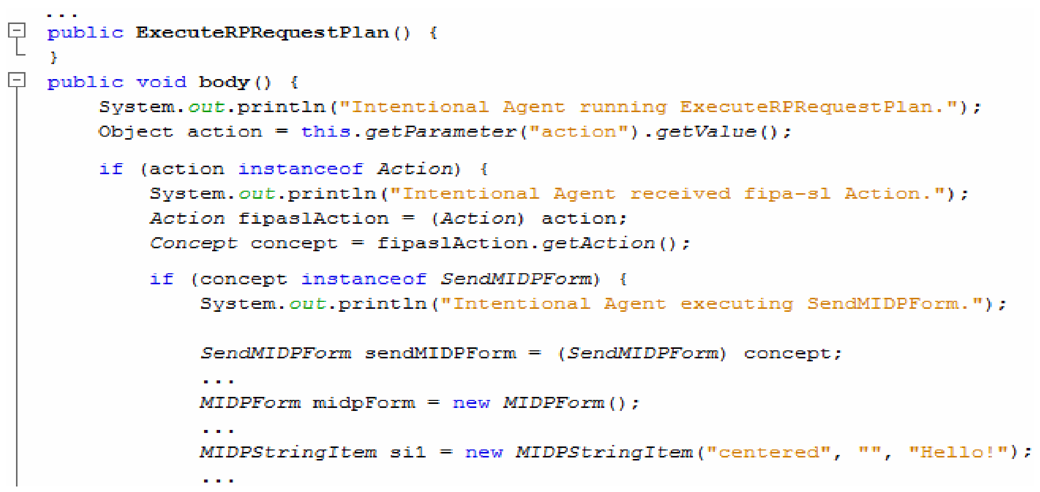
Figure 4.21 - Creating/manipulating the content expressions of *MIDP GUI Ontology* as Java objects using the *Interface Agent*

Again, in order to create and manipulate the content expressions using intentional agents, we extended the *Plan* class specified on the JADEX documentation and we also implemented the *DecideRPRequestPlan* and the *ExecuteRPRequestPlan* as plans of the *Intentional Agent*. Figures 4.22 and 4.23 respectively present code fragments of these plans.



```
    ...
    public DecideRPRequestPlan() {
    }
    public void body() {

        System.out.println("Intentional Agent running DecideRPRequestPlan.");
        ...
            if (action instanceof Action) {
                System.out.println("Intentional Agent received fipa-sl Action.")

                Action fipaslAction = (Action) action;
                Concept concept = fipaslAction.getAction();

                if (concept instanceof SendMIDPForm) {
                    this.getParameter("accept").setValue(true);
                } else {
                    this.getParameter("accept").setValue(false);
                }
            } else {
                this.getParameter("accept").setValue(false);
            }
        }
    }
}
```

Figure 4.22 - Manipulating the content expressions of *MIDP GUI Ontology* using the *Intentional Agent* (*DecideRPRequestPlan*)



```
...
public ExecuteRPRequestPlan() {
}
public void body() {
    System.out.println("Intentional Agent running ExecuteRPRequestPlan.");
    Object action = this.getParameter("action").getValue();

    if (action instanceof Action) {
        System.out.println("Intentional Agent received fipa-sl Action.");
        Action fipaslAction = (Action) action;
        Concept concept = fipaslAction.getAction();

        if (concept instanceof SendMIDPForm) {
            System.out.println("Intentional Agent executing SendMIDPForm.");

            SendMIDPForm sendMIDPForm = (SendMIDPForm) concept;
            ...
            MIDPForm midpForm = new MIDPForm();
            ...
            MIDPStringItem si1 = new MIDPStringItem("centered", "", "Hello!");
            ...
```

Figure 4.23 - Manipulating the content expressions of *MIDP GUI Ontology* using the *Intentional Agent* (*ExecuteRPRequestPlan*)

Based on the described ontological support, we also propose a dynamic interface adaptation approach for ubiquitous devices centered on intentional agents. In addition, our interface adaptation is focused on the generic components of the **G**raphical **U**ser **I**nterface (GUI), such as: *forms*, *string items*, *radio buttons* and others. Based on these components, we propose a generic ontology that describes the interface elements used to dynamically construct interfaces in our approach. Our GUI Generic Ontology is composed of common found interface elements, such as *Form*, *StringItem*, *RadioButtons*, *ImageItem*, *DateField*, *TextField*, *ChoiceGroup*, *ChoiceElement*, *List*, *ListElement*, *TextBox* and others. Moreover, it also describes components to represent specific interface elements, such as: (i) *LoginScreen* component for login information capture; (ii) *DigitalSignature* component to deal with specific devices that capture the use's digital signature at runtime, and (iii) *BiometricInformation* component for a digital finger printer, which can be captured by using biometric-based devices.

The adaptation service for MIDP devices is developed based on these interface elements and the interface elements of the GUI Generic Ontology. This service is specialized in the association between the GUI Generic Ontology and the MIDP GUI Ontology by adapting generic interface elements to allow their visualization in MIDP devices. Table 4.2 exemplifies these relationships, which are based on the similarities of the elements. Heuristics can contribute to the determination of relationships, such as the *DigitalSignature* and the *BiometricInformation* represented as *ChoiceGroups*. In order to simplify the interfaces for MIDP devices, it is necessary to consider that this kind of device does not provide powerful resources to capture the user's digital signature or even her/his biometric information. A *ChoiceGroup*, for example, can be used in this case to capture if the user does or does not agree with the service provider's rules.

Table 4.2 - GUI Elements & MIDP GUI Elements

GUI Element	MIDP GUI Element
Form	MIDPForm
ImageItem	MIDPImage
StringItem	MIDPStringItem
ChoiceGroup	MIDPChoiceGroup
…	…
LoginScreen	MIDPLoginScreen
DigitalSignature	MIDPChoiceGroup
BiometricInformation	MIDPChoiceGroup

The ubiquitous application can directly use or even reuse our interface ontological support for MIDP devices if it attends the ubiquitous application's need. However, another important step for dealing with the dynamic interface adaptation problem by considering specific needs of the ubiquitous application under analysis is the determination of the heuristics that correlate the ubiquitous application's ontology and our GUI Generic Ontology. Therefore, the interface elements of the specific ontology is associated with the interface elements of the GUI Generic Ontology, which already have an adaptation service to convert them to the interface elements of the MIDP GUI Ontology. The latter ontology is prepared to deal with interface elements even if the device is limited. Details of this kind of correlation are presented in Chapter 6 with a ubiquitous application from the dental clinic cognitive domain.

Based on the described ontological support, we propose the *Dynamic Interface Construction Building Block* (Figure 4.24). This support set has a *Reuse-Based Support* package, which is composed of our *GUI Generic Ontology* package, the *MIDP GUI Ontology* package, the *Adaptation Service*, the *Graphical User Interface (GUI)* package, and the *FIPA Standards Ontological Support* package. The application under development can just use the *GUI Generic Ontology* and/or the *MIDP GUI Ontology* as they are provided by the *Dynamic Interface Construction Building Block*. Moreover, they can be extended/instantiated to better attend the application's interface elements based on the offered interface elements. However, if the application under development have specific interface elements that do not match with the pre-defined interface elements, thus it is necessary to define these elements and establish the heuristics that associate the desire ontology and the proposed generic ontology. In addition, a new ontological java class as well as a new vocabulary must be defined. Finally, the software engineers can follow the guidelines proposed by the *Dynamic Interface Construction Building Block* to define the content language, register it, and create/manipulate it.

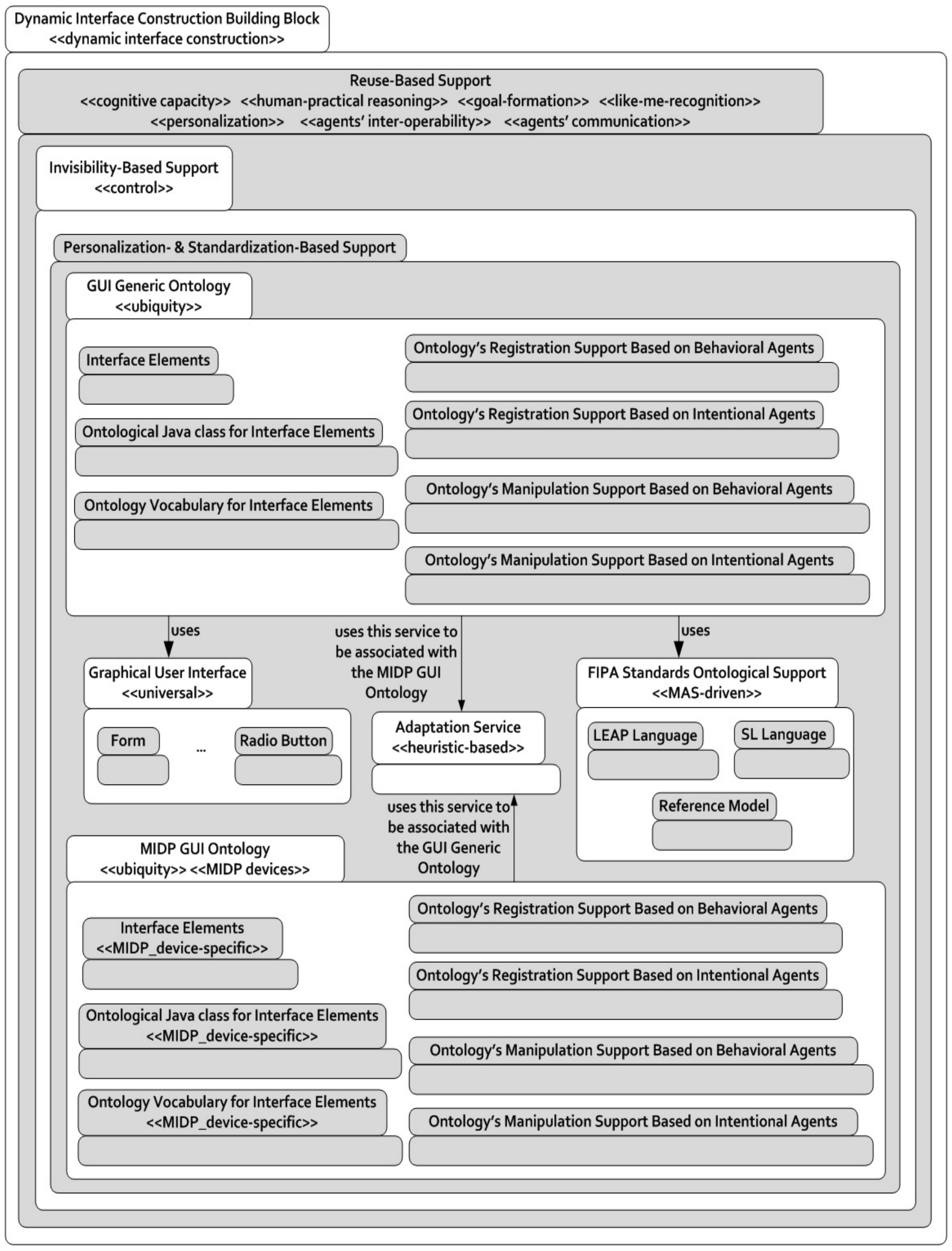

Dynamic Interface Construction Building Block <<dynamic interface construction>>
Reuse-Based Support <<cognitive capacity>> <<human-practical reasoning>> <<goal-formation>> <<like-me-recognition>> <<personalization>> <<agents' inter-operability>> <<agents' communication>>
Invisibility-Based Support <<control>>
Personalization- & Standardization-Based Support
GUI Generic Ontology <<ubiquity>>
Interface Elements
Ontological Java class for Interface Elements
Ontology Vocabulary for Interface Elements
Ontology's Registration Support Based on Behavioral Agents
Ontology's Registration Support Based on Intentional Agents
Ontology's Manipulation Support Based on Behavioral Agents
Ontology's Manipulation Support Based on Intentional Agents
uses
uses this service to be associated with the MIDP GUI Ontology
uses
Graphical User Interface <<universal>>
Form
...
Radio Button
Adaptation Service <<heuristic-based>>
FIPA Standards Ontological Support <<MAS-driven>>
LEAP Language
SL Language
Reference Model
uses this service to be associated with the GUI Generic Ontology
MIDP GUI Ontology <<ubiquity>> <<MIDP devices>>
Interface Elements <<MIDP_device-specific>>
Ontological Java class for Interface Elements <<MIDP_device-specific>>
Ontology Vocabulary for Interface Elements <<MIDP_device-specific>>
Ontology's Registration Support Based on Behavioral Agents
Ontology's Registration Support Based on Intentional Agents
Ontology's Manipulation Support Based on Behavioral Agents
Ontology's Manipulation Support Based on Intentional Agents

Figure 4.24 - *Dynamic Interface Construction Building Block* packages

4.6. Ubiquity Issues Building Bocks

Another constructed computational support – based on our empirical research of different technologies – is the *Ubiquity-Based Frameworks* that compose our *Ubiquity Issues Building Blocks*. Figure 4.25 shows the *Ubiquity-Based Frameworks* package, in which we have some frameworks to deal with specific ubiquitous concerns, such as the **I**ntentional **F**ramework for **C**ontent **A**daptation in **U**biquitous **C**omputing Systems (IFCAUC) (Serrano et al. 2008) to perform the content adaptability by considering the ubiquitous profiles and some quality criteria (e.g. security and download time). Therefore, the ubiquity-based support improves the invisibility, context-awareness and ubiquitous-profiles-awareness issues. Furthermore, it positively impacts on privacy issues by dynamically retrieving/adapting/managing the content and the services, respecting the context under analysis and the profile information, also in accordance with the users' preferences – *Personalization-Based Support*.

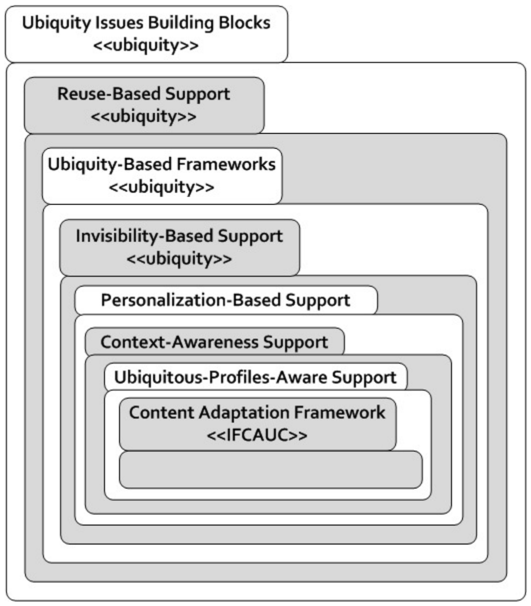

Ubiquity Issues Building Blocks <<ubiquity>>
Reuse-Based Support <<ubiquity>>
Ubiquity-Based Frameworks <<ubiquity>>
Invisibility-Based Support <<ubiquity>>
Personalization-Based Support
Context-Awareness Support
Ubiquitous-Profiles-Aware Support
Content Adaptation Framework <<IFCAUC>>

Figure 4.25 - *Ubiquity Issues Building Block* packages

In ever-changing environments, the content adaptation is really important to improve or even to guarantee user satisfaction by considering the service omnipresence supported by different devices. The content adaptation can be described as a process in which the requested contents are adapted according to specific profiles. Therefore, we cannot think about content adaptation without thinking about profiles. Profiles are the main key of an appropriate content adaptation. This notion is intrinsic to Ubiquitous Computing and commonly applied to the ubiquitous applications development (W3C - CC/PP 2011; Berhe et al. 2004).

The main ubiquitous profiles are: (i) the user profile, which represents the user preferences and personal data; (ii) the device profile, which contains the device features (e.g. resolution, memory and processing capacities); (iii) the network profile, which stores the network specifications (e.g. bandwidth); (iv) the content profile, which represents the content characteristics (e.g. we can consider the size to download and the type as some characteristics for a media content); and (v) the contract profile, which contains the contract information that is established between the service provider and the final user.

There are three ways to perform content adaptation in ubiquitous applications: (i) inside the device, in which it is important to deal with the device memory and processing capacities; (ii) in the application server, in which it is relevant to consider a possible server overload; and (iii) in a dedicated server, in which we must consider a specific support to avoid problems with the security of information that is exchange between the application server and the dedicated server.

Additionally, the context information can be described as dynamic or static. The dynamic information (e.g. requested content and network specifications) must be obtained during the adaptation process at runtime. The static information (e.g. user personal data) can previously be defined and stored for further consultation. However, the proliferation of different devices combined with the necessity of the context-aware service personalization emphasizes the importance of technological support to deal with dynamic information. Our approach tries to fill this gap by providing a dynamic content adaptation driven by intentional agents to adequately satisfy the user’s expectations.

Moreover, our approach demands different content adaptations. Therefore, we also classified them into five main categories: (i) adaptation based on resizing, to adapt the content according to the device screen resolution; (ii) adaptation based on transcoding, to transcode the content from one format to another; (iii) adaptation based on reduction, to adapt the content using data compression; (iv) adaptation based on replacement, to replace a sequence with still frames, which are combined to form a slide show; and (v) adaptation based on integration, to adapt the content using service composition. For example, a video can be obtained by combining different image frames with the corresponding audio.

Summarizing the proposed content adaptability process (Figure 4.26): (i) the request is performed by the client using her/his device; (ii) the device is integrated by the autonomous entities with the platform container (main container or other); (iii) the agents (e.g. Interface Agent, Initiator Agent, AMS Agent, DF Agent, Mobile Agent and Adapter Agent) collaborate to achieve the client's goal (e.g. download a content); (iv) the desired content as well as its service provider is identified by considering specific quality criteria (e.g. security and price), which are specified and analyzed by the agents at runtime using fuzzy variables, fuzzy sets and fuzzy conditional rules; (v) the adaptability need is detected according to the ubiquitous profiles, the agents' belief base, and the context under analysis; (vi) the content is adapted in a dedicated server by performing specific content adaptation techniques (e.g. resizing) and/or the combination of them (e.g. resizing and transcoding); and (vii) the adapted content is provided to the user on her/his device by considering, among other ubiquitous profiles information, the device's screen resolution, the device's accepted colors and the user's preferences. As follows, some details of this process are presented.

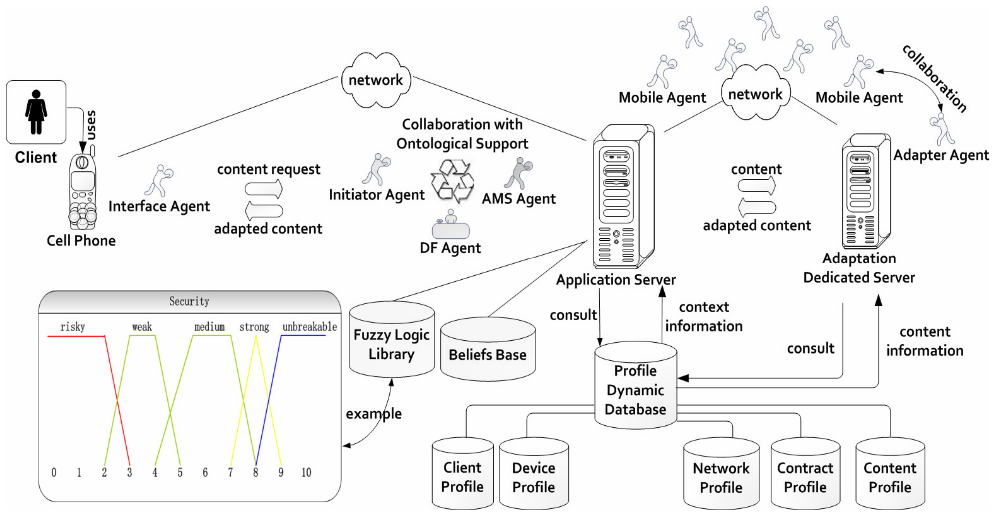


Figure 4.26 - The proposed content adaptability process

In the described process, the user's device is identified at runtime by using different properties depending on of the device's platform:

- **jse devices** – are based on the **J**ava **S**tandard **E**dition platform, such as laptops and desktop PCs. This platform is a widely used platform for programming in the Java language. In other words, it consists of a virtual machine, which must be used to run Java-based programs by allowing file systems, networks and graphical interfaces from within those programs.

- **Java Personal Edition devices** – are based on the Personal Java platform, such as modern Smatphones (e.g. Blackberry). This platform is a Java edition for mobile and embedded systems based on the **Windows E**mbedded **C**ompact (Windows CE). The Windows CE represents an operating system specifically developed by the Microsoft for minimalist computers and embedded systems.
- **jme devices** – are based on the Java Microedition platform, such as the MIDP devices. This platform is a collection of Java APIs, which is defined by the **J**ava **C**ommunity **P**rocess (JCP). It provides resources for the development of embedded systems and applications, which run inside a device with a specific purpose. Normally, this device is limited in comparison with the jse- and Personal Java-based devices. Therefore, jme devices are a concern in Ubiquitous Computing, by emphasizing the importance of the content adaptability.

Now, we will describe how to identify the device at runtime. For devices that run jse, the *public static final boolean* is jse. Therefore, we use: *String property = System.getProperty("java. Runtime.name")*. On one hand, *if (property! = null)*, then the platform is jse, which means that it is a powerful machine, capable, for example, of receiving images with large resolution. On the other hand, if the device is not jse, then it is possible to be a Personal Java or a MIDP device. To check if the device is running Personal Java the detecting program asks if the system *property os.name* contains the value Windows CE. As Windows CE only runs Personal Java the platform will be Personal Java. Moreover, the Personal Java devices include the iPAQ (high- spec PDA). If the *string* is anything else or null then the platform is not Personal Java. Finally, to check if the device is running MIDP the detecting program identifies it by using the microedition properties (Microedition Properties 2005) (e.g. *microedition.platform*, *microedition.profiles* and *microedition.configuration*). The detecting program identifies the platform details, which means the proper java platform type and the device's model. Based on the model, it is possible to retrieve its profile from the dynamic database and determine its limitations in terms of memory capacity, processing capacity, resolution and other features at runtime.

Continuing our adaptation process, the device is integrated to the agents' platform by an Interface Agent. This agent also requests the creation and registration of an Initiator Agent – i.e. a personal agent of the client – for respectively the AMS Agent and the DF Agent. The Interface Agent intermediates the Initiator Agent and the client communication. The agents' communication and inter-operability is facilitated by an ontological support (previously presented) that describes, for example, the interface elements. Based on this ontology, the agent knows how to adapt the content and all forms that will be exchanged during the adaptation process. Moreover, the ontology describes how to present this content as well as the forms from the device that the client is using at the moment of the request. Therefore, it is also necessary to consult the profiles by using a **D**ata **A**ccess **O**bject (DAO) (DAO 2011) – a data persistence pattern. For example, to consult the user profile in the dynamic database, the Initiator Agent uses a data access object and the **H**ibernate **Q**uery **L**anguage (HQL) (HQL 2011) – a powerful query language, which is similar to SQL. However, it is "*fully object-oriented and understands notions like inheritance, polymorphism and associatio*n (HQL 2011)."

Furthermore, the Initiator Agent requests the creation and registration of a Mobile Agent for respectively the AMS Agent and the DF Agent. The Mobile Agent receives the client and the device information to adapt the content in a dedicated server. The Mobile Agent migrates, performs the adaptation with the Adapter Agent by respecting the client's preferences and the device features, returns to the application's server and sends the adapted content to the Initiator Agent. This latter agent sends this content to the Interface Agent, which performs the visualization of the adapted content to the client from her/his device. It is important to notice that the client is not previously associated with a specific device and its features are constantly updated based on the WURFL Repository (WURFL 2011a; WURFL 2011b) (presented on Section 4.7). Therefore, the client can change the device, perform another request and the MAS-oriented application is able to identify the device at runtime, without disturbing the client or even distracting her/him, as idealized by Mark Weiser's vision – the complexity invisibility need centered on Calm Technology.

4.7. Dynamic Database Building Block

Another technological contribution of our approach to improve the reuse in ubiquitous scenarios is our *Dynamic Database Building Block* centered on the Type-Square Architecture, the WURFL Repository and a Persistence Framework.

4.7.1. Type-Square Architecture

In ever-changing contexts, the user desires to change her/his preferences and device anywhere and at any time. Contributing to this difficult scenario, the devices are in constant evolution by following novel technologies (e.g. yesterday the devices contained a CD reader/writer, today they have a DVD reader/writer and tomorrow all of them will have the blue-ray reader/writer). Moreover, heterogeneous devices (e.g. mobile, small, just-call-phone, limited or powerful) enter and leave different intelligent environments. The heterogeneity, the large number of users and devices and the constant evolution in terms of user preferences and device features are intrinsic in ubiquitous scenarios. Therefore, they require adequate support to adapt and quickly change the user profile and the device profile according to each user's requirement. This usually is achieved by storing user preferences, device features and other dynamic data (e.g. network specifications and contract information) in a dynamic database.

Dynamic database is a new kind of value-based database (e.g. relational database), in which tables, fields and values can be created, manipulated and excluded wherever changes need to be made with immediate (but controlled) effects on the system interpreting it. Moreover, the database relationships as well as the location of related records are respectively specified and determined at runtime. In order to improve the development of a dynamic database, the use of a specific architecture that can dynamically adapt to new contexts at runtime is appropriate. This kind of architecture is sometimes called a reflective-architecture or a meta-architecture. Our approach is centered on a particular meta-architecture – Type-Square Architecture focused on the Type-Object Pattern – which was first proposed by (Yoder et al. 2001), as illustrated in Figure 4.27.

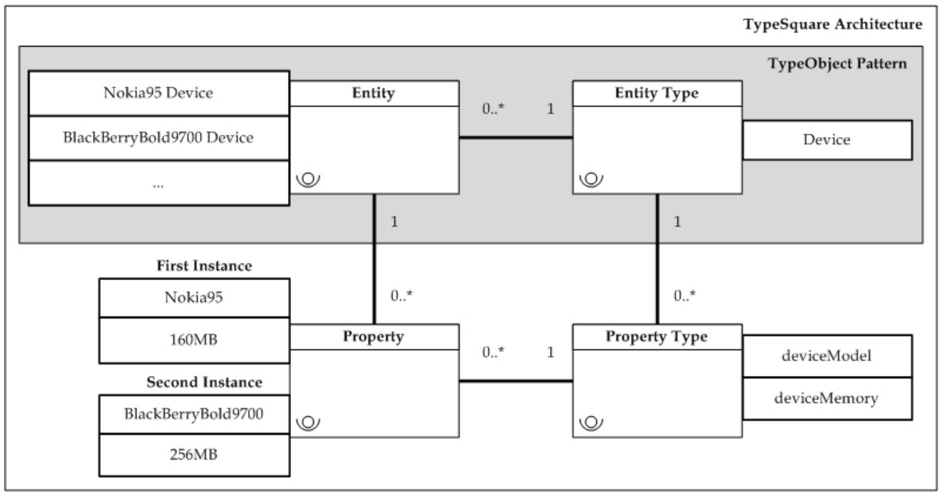


Figure 4.27 – *Type-Square* architecture (adapted from (Yoder et al. 2001))

In Figure 4.27, the Entity-Type represents the classes and the Entity represents the class instances. As a simple example, we have the Entity-Type "*Device*" and the Entity "*Nokia95 Device.*" "*Nokia95 Device*" is an instance of "*Device*." Moreover, the Entity is associated with a specific Entity-Type and the Entity-Type can be associated with zero or various Entities. Thus, the cardinality in the first way is one to one (1..1) and in the opposite way is one to zero or more (1 to 0..*). We can have different devices – e.g. "*BlackBerryBold9700 Device*," "*NokiaN86 Device*," "*MotorolaWX390 Device*" and "*SonyEricssonXperia-X10 Device*." Each of them (an Entity) is an instance of "*Device*" (an Entity-Type).

The "*Device*" Entity-Type can have different Properties-Types, such as: "*deviceModel*," "*deviceMemory*," "*deviceScreenSize*," "*deviceBattery*," and "*deviceOperatingSystem*." Focused on this idea, the architecture stores the properties' values of a specific Entity (e.g. "*Nokia95 Device*") as Properties. Thus, for the "*Nokia95 Device*" Entity, the "*deviceModel*" is "*Nokia95*" and the "*deviceMemory*" is "*160MB*". The architecture also specifies other important associations between: (i) Property and Property-Type – one Property must be associated with only one Property-Type – e.g. the "*Nokia95*" Property is only associated with the "*deviceModel*" Property-Type and (ii) Property-Type and Property – one Property-Type can be associated with zero or more Properties – e.g. the "*deviceMemory*" Property-Type is associated with the "*160MB*" and "*256MB*" Properties.

We extended this architecture model to better attend our needs by improving the inheritance concept in multiple levels. In Figure 4.28 the Entity-Property represents a new class/table in our Dynamic Database model that is created based on the association between Entity and Property, which cardinality is ***..***. It means that an Entity can be associated with zero or more (**0..***) Property and a Property can be associated with zero or more (**0..***) Entity.

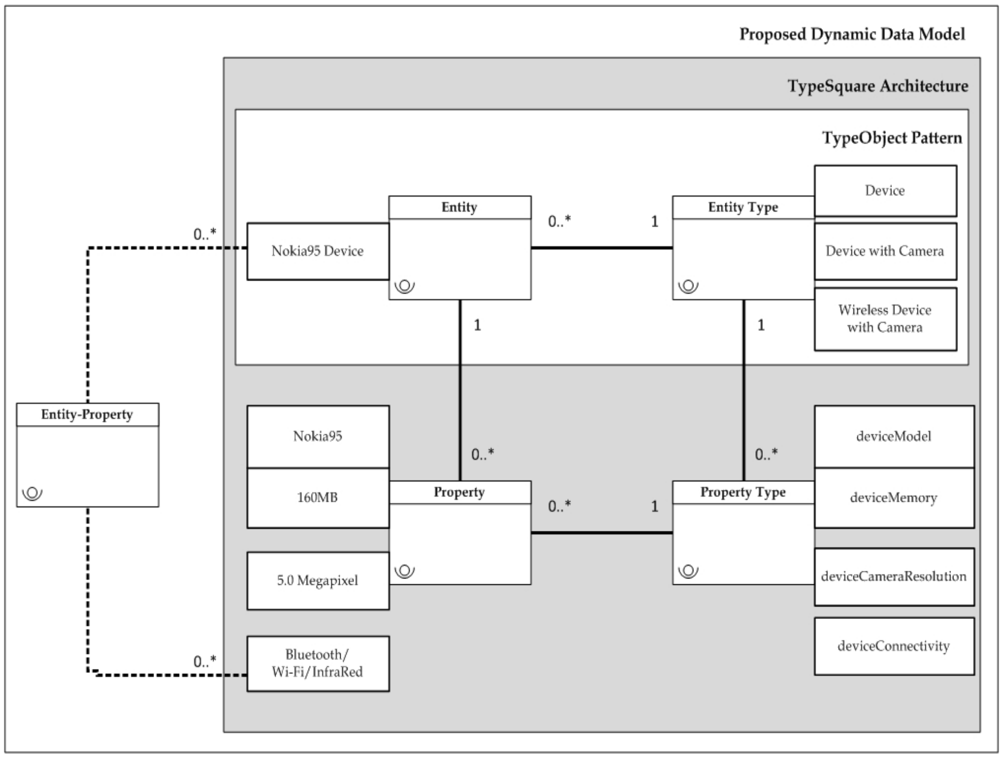


Figure 4.28 - Dynamic database architecture (first example)

One of our main purposes is to deal with contexts, such as: "*A specific device (Entity) is an instance of Device (Entity-Type). Thus, this specific device (e.g. Nokia95 Device) contains the Device's properties. Moreover, this same specific device (Entity) is an instance of Device with Camera (other Entity-Type). Thus, this specific device (e.g. Nokia95 Device) also contains the Device with Camera's properties. However, the Device with Camera is a Device! Thus, a Device with Camera inherits the Device's properties.*"

We have the inheritance concept in the context presented before and the Type-Square Architecture does not directly deal with this kind of context. Our Dynamic Database allows dynamically defining one or more levels of inheritance: a *"Specific Device"* (an Entity) is an instance of *"Device with Camera"* (an Entity-Type), which is a *"Device"* (an Entity-Type→First-Level-Of-Inheritance).

Moreover, a *"Specific Device"* (an Entity) is an instance of *"Wireless Device with Camera"* (an Entity-Type), which is a *"Device with Camera"* (an Entity-Type→ Second-Level-Of-Inheritance). Only to illustrate, consider the example previously presented in Figure 4.28 and the explanation as follows:

- Instantiation: As the *"Nokia95 Device"* is an instance of *"Wireless Device with Camera,"* it contains the *"deviceConnectivity"* Property, which value is *"Bluetooth/Wi-Fi/InfraRed";*
- First-Level-Of-Inheritance: As a *"Wireless Device with Camera"* is a *"Device with Camera,"* it also inherits the *"Device with Camera"* Property-Types (e.g. *"deviceCameraResolution"*). We represent this relationship as a special Property-Type called *"SUPER,"* which type is *"Device with Camera."* Thus, the *"Nokia95 Device,"* as an instance of *"Wireless Device with Camera,"* will contain two Properties-Types *("super"* and *"deviceConnectivity"*), which values are respectively an object of *"Device with Camera"* (in which *"deviceCameraResolution"* Property-Type is associated with the value *"5.0 Megapixel"*) and *"Bluetooth/Wi-Fi/InfraRed"; and*
- Second-Level-Of-Inheritance: As a *"Device with Camera"* is a *"Device,"* it also inherits the *"Device"* Property-Types (e.g. *"deviceModel"* and *"deviceMemory"*). We represent it as a Property-Type *"SUPER,"* which type is *"Device."* Thus, the object *"SUPER"* of the *"Nokia95 Device"* will contain two Properties-Types *("super"* and *"deviceCameraResolution"*), which values are respectively an object of *"Device"* (in which *"deviceModel"* and *"deviceMemory"* Properties-Types are associated with the values *"Nokia95"* and *"160MB"*) and *"5.0 Megapixel".*

Another context that we try to deal with is: "*A specific device (Entity) is an instance of Device (Entity-Type). Thus, this specific device (e.g. Nokia95 Device) contains the Device's properties (deviceModel, deviceMemory and deviceBattery). In this context, a Device (Entity-Type) has battery as Property-Type. BUT Battery is an Entity-Type, which has batteryType and batteryCapacity as Properties-Types. It means that Device (Entity-Type) is associated with Battery (another Entity-Type).*"

We have a classical association in the context presented before and again the Type-Square Architecture does not directly deal with this kind of context. Our

Dynamic Database proposes a various-to-various association between the *"Device"* Entity-Type and the *"Battery"* Entity-Type, represented by the cardinality **0..*** to **0..*** and the new class/table Entity-Property (see Figure 4.29) .

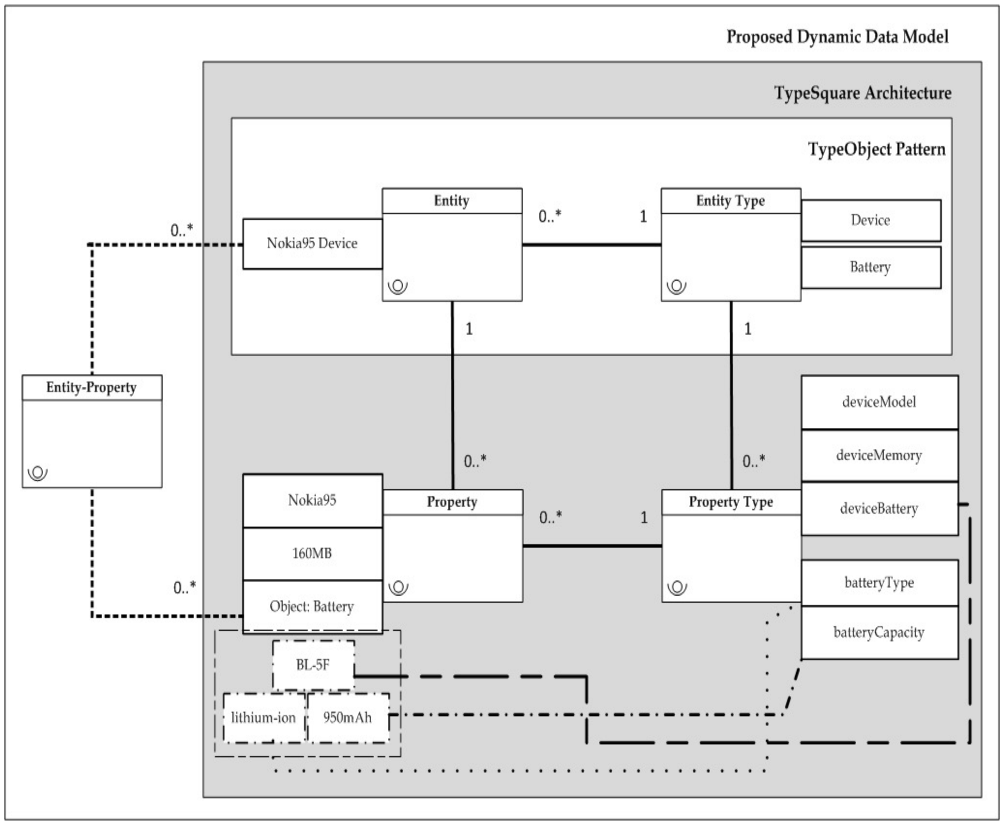


Figure 4.29 - Dynamic database architecture (second example)

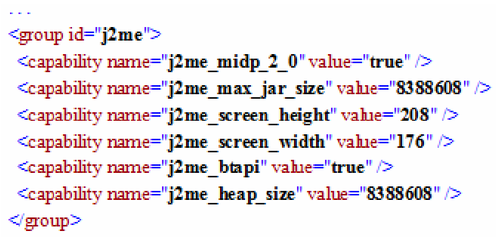
For example:

- Instantiation: As the *"Nokia95 Device"* is an instance of *"Device,"* it contains the *"deviceModel," "deviceMemory"* and *"deviceBattery"* as its Properties, which values are respectively *"Nokia95," "160MB"* and an object *"BL-5F."*
- Association: the object *"BL-5F"* is a *"Battery."* It is represented as an association between *"Battery"* and *"Device."* Thus, the *"Nokia95 Device"* also contains *"batteryType"* and *"batteryCapacity"* as its Properties, which values are respectively *"lithium-ion"* and *"950mAh."*

4.7.2.
WURFL Repository & Persistence Framework

In order to deal with the device technological evolution and other ever-changing issues, we use a specific repository – named WURFL (**W**ireless **U**niversal

Resource **F**i**L**e) and proposed by an open-source project (WURFL 2011a) – to constantly update the device profile in our dynamic database. In short, the WURFL is an XML configuration file (Figure 4.30) of device capability information. In this context, capability means the ability to support certain features (e.g. image formats, memory capacity, processing capacity, mark-ups, screen resolution and screen colors). The repository contains over 500 capabilities for each device that are divided into 30 groups, whose complete listing is available on the WURFL documentation page (WURFL 2011b).



```
...
<group id="j2me">
  <capability name="j2me_midp_2_0" value="true" />
  <capability name="j2me_max_jar_size" value="8388608" />
  <capability name="j2me_screen_height" value="208" />
  <capability name="j2me_screen_width" value="176" />
  <capability name="j2me_btapi" value="true" />
  <capability name="j2me_heap_size" value="8388608" />
</group>
```

Figure 4.30 - Code fragment of the WURFL XML file

The profiles depend on the information acquisition, which can be performed at runtime or previously obtained. Some examples of the acquisition process at runtime are: (i) in systems that involve the user’s navigation, the information about the user’s interests can be elicited during this navigation by following the user’s accesses, (ii) the user’s information can also be acquired by using her/his registration, (iii) the information about the content (e.g. its format and its resolution) can be acquired with the proper content provider or by using a software that extracts the content file’s properties at runtime and stores them into the Content Profile, and (iv) when the device is identified (e.g. by applying the jse or the microedition properties or by using an URL (**U**niform **R**esource **L**ocation (Berners-Lee et al. 1994)), it is possible to recover its features – stored in the device profile, which are constantly updated by using the WURFL Repository – at runtime. Based on the profile information, the content adaptation can be dynamically performed to adequately satisfy the user’s needs.

A *Persistence Framework* – e.g. the *Hibernate Framework* (Hibernate 2011) – is used to manipulate the *Dynamic Data Model* centered on the ubiquitous profiles. Figure 4.31 illustrates the proposed *Dynamic Database Building Block* with a *Reuse-Based Support* package mainly composed of the *Persistence Framework* package and the *Personalization-Based Support* package focused on the ubiquitous profiles, the extended Type-Square Architecture and the WURFL Repository.

The *Persistence Framework* package combined with the *Dynamic Data Model* package positively impacts on the invisibility. Moreover, as previously explained, the proposed data model is based on a flexible entity-model – the Type-Square Architecture – to allow the storage, retrieving and exclusion of ubiquitous profiles information at runtime. Concluding the presentation of the *Dynamic Database Building Block*, the *Personalization-Based Support* helped us with regard to the constant data management of different profiles by facilitating, for example, the acquisition of the user preferences "on the fly" in order to improve – among other contributions – the user satisfaction.

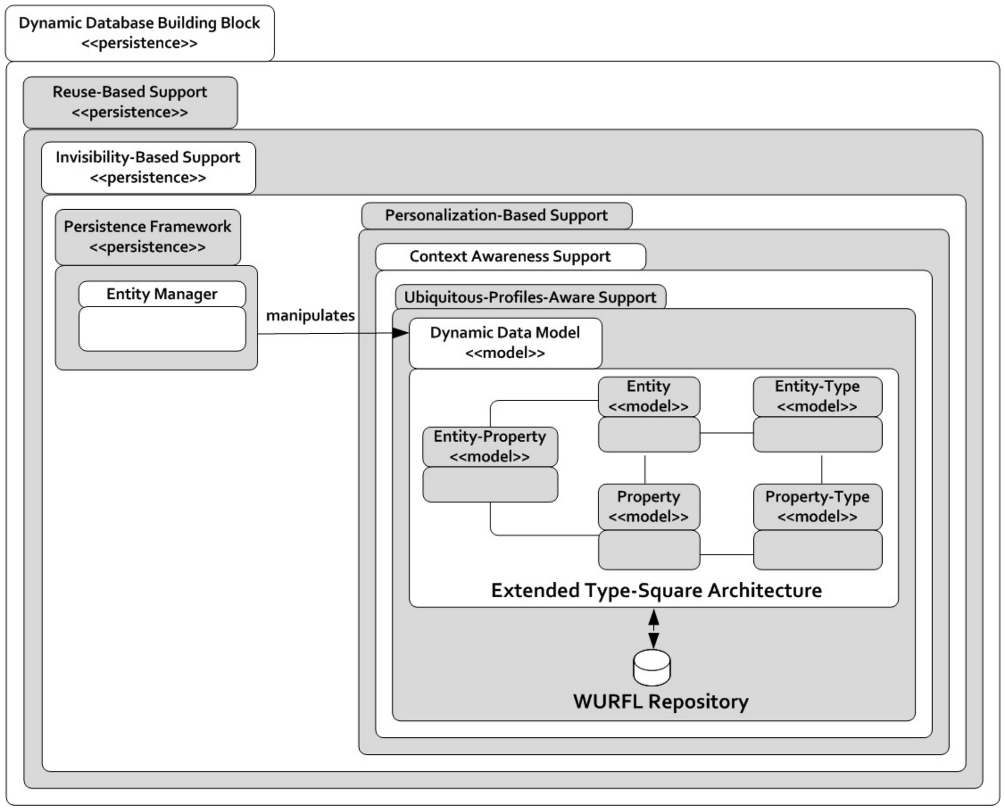


Figure 4.31 – *Dynamic Database Building Block* packages

4.8. Closing Remarks

This Chapter presented the main technological support sets – i.e. building blocks – developed by our approach – the *Domain Engineering of Ubiquitous Applications* – for reuse in order to systematically and incrementally develop intentional-MAS-driven ubiquitous applications. The *Intentional Modeling Building Block* centered on the i* Framework is used to model ubiquitous applications based on the intentionality concept. The *NFR Catalogue Building Block* focused on the NFR Framework provides models of non-functional ubiquitous requirements, with their interdependencies and operationalizations. The *Integration Building Block* based on the JADE-LEAP Platform offers execution modes to deal with different devices and resources to integrate distributed smart-spaces with the MAS platform. The *Intentional Agents' Reasoning Building Block* focused on the JADEX Framework provides a reasoning engine based on the BDI model and the capability concept to improve the agents' cognitive ability. The instantiation of the *Fuzzy Logic Library Support* improves the *Intentional Agents' Reasoning Building Block* by allowing intentional agents to deal with non-functional requirements (e.g. security, price and other quality criteria) at runtime. The *Ubiquity Issues Building Blocks* based on Ubiquity-Based Frameworks deal with specific ubiquitous concerns, such as the IFCAUC for the content adaptation issue. The *Dynamic Database Building Block* centered on the Type-Square Architecture, the WURFL Repository and a Persistence Framework is another contribution of our approach. The Type-Square is basically a meta-architecture to improve the data management "on the fly". The WURFL Repository is an international repository that evolves over time by updating the device profile and following the technological trends. The Persistence Framework – in our case, the Hibernate Framework – is used to manipulate the proposed dynamic data model.

In the next Chapter, we will discuss about our reuse-oriented proposal centered on the building blocks developed from the *Domain Engineering of Ubiquitous Applications* (presented in this Chapter) and the *Ubiquitous Application Engineering*. In the *Ubiquitous Application Engineering,* we propose the reuse of these building blocks (organized in different packages) to facilitate the incremental and systematic development of ubiquitous applications by offering an intentional-MAS-driven suitable support to deal with the main concerns commonly found in these applications, such as: the complexity invisibility, the integration need, the content adaptation, the context awareness, the ever-changing conditions, and other intrinsic issues.