HAL
open science

Qualité de l'interaction homme machine : interfaces auto-explicatives par ingénierie dirigée par les modèles

Alfonso Garcia Frey

To cite this version:

Alfonso Garcia Frey. Qualité de l'interaction homme machine : interfaces auto-explicatives par ingénierie dirigée par les modèles. Other [cs.OH]. Université de Grenoble, 2013. English. NNT : 2013GRENM015 . tel-01138082

HAL Id: tel-01138082

https://theses.hal.science/tel-01138082

Submitted on 1 Apr 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Alfonso García Frey

Thèse dirigée par **Gaëlle Calvary**
et codirigée par **Sophie Dupuy Chessa**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Quality of Human-Computer Interaction: Self-Explanatory User Interfaces by Model-Driven Engineering

Thèse soutenue publiquement le **03 Juillet 2013**,
devant le jury composé de :

Mrs. Karin Coninx
Professeur à l'Université de Hasselt, Rapporteur
Mr. Jean Vanderdonckt
Professeur à Université catholique de Louvain, Rapporteur
Mr. James Crowley
Professeur à Grenoble INP, Examinateur
Mr. Eric Dubois
Professeur, Directeur au Centre de Recherche Public Henri Tudor, Examinateur
Mr. Víctor Manuel López Jaquero
Professeur à l'Universidad de Castilla la Mancha, Examinateur
Mr. Philippe Renevier-Gonin
Maître de conférences à l'Université Nice Sophia Antipolis, Examinateur
Mrs. Gaëlle Calvary
Professeur à Grenoble INP, Directrice de thèse
Mrs. Sophie Dupuy Chessa
Maître de conférences à l'Université Pierre-Mendès-France, Co-Directrice de thèse

To Ioana. You already know why.

Acknowledgements

Firstly I would like to thank my supervisors, Gaëlle Calvary and Sophie Dupuy Chessa. Gaëlle Calvary introduced me to the world of HCI while I was a Master student and gave me the opportunity to enter the HCI research team of the Laboratoire d'Informatique de Grenoble. I am very grateful to Gaëlle Calvary and Sophie Dupuy Chessa for being so supportive, and for giving me the opportunity to work with them. Thanks to them I was able to participate to many conferences and to meet so many other great researchers from all over the world. It is unlikely I would have come this far without their guidance, support and patience through all these years. Thank you very much.

I would also like to thank every member of the HCI group for their collaboration, support, and discussions; special mention should be made of Eric Céret for his close collaboration. Also big thanks to Joëlle Coutaz and Alexandre Demeure for the fructiferous discussions on modeling and meta-modeling.

During this research I met incredible people, among these, Anke Dittmar (University of Rostock). Working with Anke was very enriching as she is a great researcher, colleague, and very kind person. Thank you.

I am very thankful to the amazing members of the UsiXML consortium. I am glad to have met such outstanding researchers that have made the UsiXML language possible. The UsiXML project was funded by the ITEA2.

I extend my sincere thanks to all the members of the jury. It is an honour for me to have this high quality panel with such valuable and special members. Thank you very much.

Last but not least, a final thank to my family, my friends, and my girlfriend, for their support and encouragement over the years.

Abstract

In Human-Computer Interaction, quality is an utopia. Despite all the design efforts, there are always uses and situations for which the user interface is not perfect. This thesis investigates self-explanatory user interfaces for improving the quality perceived by end users. The approach follows the principles of model-driven engineering. It consists in keeping the design models at runtime so that to dynamically enrich the user interface with a set of possible questions and answers. The questions are related to usage (for instance, "What's the purpose of this button?", "Why is this action not possible"?) as well as to design rationale (for instance, "Why are the items not alphabetically ordered?").

This thesis proposes a software infrastructure UsiExplain based on the UsiXML metamodels. An evaluation conducted on a case study related to a car shopping website confirms that the approach is relevant especially for usage questions. Design rationale will be further explored in the future.

Contents

1 Introduction **1**

1.1 Research Problem and Motivation 1

1.2 Thesis Approach 5

1.3 Working Hypothesis and Thesis Statement 6

1.4 Research Questions 7

1.5 Dissertation Structure 8

2 State of the Art **11**

2.1 What is an Explanation 12

2.1.1 Theory of Explanation in Philosophy of Science 12

2.1.2 Erotetic Logic: Subject and Request 13

2.1.3 Structural Explanations 14

2.2 Approaches 15

2.2.1 Expert Systems 15

2.2.1.1 General Principles 15

2.2.1.2 Knowledge-Based Systems 16

2.2.1.3 Intelligent Agents and Cooperative Support 17

2.2.2 Question Answering Systems 20

2.2.2.1 General Principles 20

2.2.2.2 Question Types in Question Answering Systems 22

2.2.3 Model-based explanations 28

2.2.3.1 Task Models 29

2.2.3.2 Behaviour Models 30

2.2.4 Social-Network Based Systems 31
2.2.5 Personal assistants 32
2.2.6 Recommender Systems 33
2.2.7 Desktop facilities 33
2.2.8 Avatars 34
2.3 Analysis of the approaches 35
2.3.1 Criteria and their application 35
2.3.1.1 Coverage of Questions 35
2.3.1.2 Quality of Answers 36
2.3.1.3 Cost 38
2.3.2 Conclusion 40
2.4 Focus on Model-Based Approaches 42
2.4.1 The QAP Problem Space 42
2.4.1.1 Questions 44
2.4.1.2 Answers 46
2.4.1.3 Properties 49
2.4.2 Reading the QAP Problem Space: Values of the Axes 51
2.4.3 QAP Problem Space and Related Work 52
2.4.3.1 Crystal System 52
2.4.3.2 PervasiveCrystal System 54
2.4.3.3 Cartoonist System 57
2.4.3.4 Intelligibility Toolkit 59
2.4.4 Overlapping Analysis 62
2.5 Synthesis 64

3 Foundations **67**
3.1 Model-Driven Initiatives 68
3.1.1 Model-Driven Initiatives: A Brief History 68
3.1.2 Model-Driven Architecture 69

3.1.2.1 Models 70
3.1.2.2 Meta-Models 71
3.1.2.3 Meta-Meta-Models 72
3.1.3 Four-layers architecture 73
3.1.3.1 Model Transformations 74
3.1.3.2 MDA Models 76
3.1.3.3 The MDA Process 77
3.1.4 Models, Meta-Models, and Meta-Meta-Models 77
3.1.5 Model-Driven Development 78
3.1.6 Model-Driven Engineering 79
3.1.7 Model-Based Engineering 79
3.2 Model-Driven Engineering of User Interfaces 80
3.2.1 The Cameleon Reference Framework 80
3.2.2 Levels of Abstraction 82
3.2.3 The UsiXML Language 85
3.3 Quality Models 87
3.3.1 Quality Model Definition 87
3.3.1.1 McCall's Software Quality Model 88
3.3.1.2 Boehm's Quality Model 88
3.3.1.3 Dromey's Quality Model 90
3.3.2 ISO Standards 91
3.3.3 QUIM Model 94
3.3.4 Finne's Quality Meta-Model for Information Systems 96
3.3.5 Ergonomic Guides 98
3.3.5.1 Bastien and Scapin 98
3.3.5.2 Vanderdonkt's Ergonomic Guide 99
3.4 Synthesis 99

4 Self-Explanatory User Interfaces **101**

4.1 Introduction 102

4.2 Gulf of Quality 104

4.3 Design Principles 107

4.3.1 Help Systems Functionality 107

4.3.2 The Global Approach 109

4.3.3 Design principles 111

4.3.3.1 Building the UI of the Help System 112

4.3.3.2 Building the UI of the application 113

4.3.3.3 Adding support for computing help 113

4.3.3.4 Weaving the UIs 115

4.4 Explanation Strategies 118

4.4.1 Determining the Appropriate Explanation Strategy 118

4.4.2 Procedural Questions - How 121

4.4.2.1 Generating Questions 121

4.4.2.2 Retrieving Information 122

4.4.2.3 Providing Support 124

4.4.3 Purpose/Functional Questions - What is it for 125

4.4.3.1 Generating Questions 126

4.4.3.2 Retrieving Information 127

4.4.3.3 Providing Support 128

4.4.4 Localization Questions - Where 130

4.4.4.1 Generating Questions 130

4.4.4.2 Retrieving Information 133

4.4.4.3 Providing Support 133

4.4.5 Availability Questions - What Can I Do Now 134

4.4.5.1 Generating Questions 135

4.4.5.2 Retrieving Information 135

4.4.5.3 Providing Support 137
4.4.6 Behavioural questions - Why I can't 138
4.4.6.1 Generating Questions 139
4.4.6.2 Retrieving Information 140
4.4.6.3 Providing Support 142
4.5 Synthesis 143

5 Design Rationale Questions 145

5.1 Design Rationale 147
5.2 QUIMERA: The Quality Meta-Model 149
5.2.1 Principles 150
5.2.2 Quality Perspectives 152
5.2.3 The Quality Meta-Model 154
5.2.4 Global Quality vs Local Quality 155
5.2.5 Quality Models: Instantiation Examples 157
5.2.5.1 A quality model covering the ergonomic criteria in HCI 157
5.2.5.2 Application to the evaluation of a design method 159
5.2.6 How to build a Quality Model 159
5.3 Design Rationale and Quality 162
5.3.1 Putting the Pieces Together 162
5.3.2 Advantages and Limitations 164
5.4 Explanation Strategy 166
5.4.1 Generating Questions 166
5.4.2 Retrieving Information 166
5.4.3 Providing Support 167
5.5 Synthesis 168

6 Self-Explanatory UIs in Action: Implementation and Evaluation 171

6.1 UsiExplain: A Model-Based Generic Architecture 172
6.2 UsiComp: a Services Oriented Framework 176
6.2.1 Services and OSGi 176
6.2.2 UsiComp Overview 178
6.2.2.1 Design Module 178
6.2.2.2 Meta-Models 180
6.2.2.3 Transformations 181
6.2.2.4 Runtime Module 182
6.2.2.5 Code Generation 183
6.2.2.6 Extension abilities 184
6.3 Relationship between UsiExplain and UsiComp 184
6.4 UsiCars: an UsiExplain Based Prototype 186
6.4.1 Prototype Description 187
6.4.2 Self-explanatory dialogue 189
6.5 Evaluation 190
6.5.1 Participants 190
6.5.2 Evaluation Protocol 191
6.5.3 Tasks 192
6.6 Qualitative analysis 193
6.6.1 Findings 194
6.6.2 Unsupported types of questions 196
6.6.3 Usability Suggestions and Improvements 197
6.6.4 Limitations of the experiment 198
6.7 Synthesis 198

7 Conclusions and Future Directions 201

7.1 Summary of the Contributions 202
7.2 Answers to Research Questions 203
7.3 Advantages of the approach 204

7.3.1 Properties of the Approach 204
7.3.1.1 Unification of question types 204
7.3.1.2 Introspection 204
7.3.1.3 Flexibility for Weaving 205
7.3.1.4 Distributability 205
7.3.1.5 Reusability 206
7.3.1.6 Customization 206
7.3.1.7 Open Approach 206
7.3.2 Proposed Solution on the QAP Problem Space 207
7.4 Limitations of the Approach 209
7.4.1 Usability improvements 209
7.4.2 Semantic Information 210
7.4.3 Scalability 210
7.5 Future Work 210
7.6 Short Term Perspectives 211
7.6.1 Usability Improvements 211
7.6.2 Interaction Techniques 211
7.6.3 Closing the Loop 212
7.7 Long Term Perspectives 212
7.7.1 Initiative Axis 212
7.7.2 Quality guided development and evaluation 213
7.7.3 Supporting New Question Types 213
7.7.4 Supporting New Sources of Knowledge 214
7.7.5 Design Rationale for Learning / End-User Programming 215

Appendices **217**

A Specification of the Quimera Quality Meta-Model **218**

B Meta-Models **224**

B.1 Tasks 224

B.2 Domain 226

B.3 AUI 226

B.4 CUI 226

B.5 Mapping 227

B.6 QOC 230

C Contributory Papers **232**

List of Figures **234**

Glossary **259**

Acronyms **261**

1

Introduction

> *The last thing one knows when writing a book is what to put first.*

Blaise Pascal,

1.1 Research Problem and Motivation

A recurrent problem in interactive systems is that users may require assistance while interacting with a User Interface (UI). As stated in [100] "Modern applications such as Microsoft Word have many automatic features and hidden dependencies that are frequently helpful but can be mysterious to both novice and expert users".

One of the classic guidelines for user interface design [104] is to have "visibility of system status" to "keep users informed about what is going on". And yet, as noticed in [100], "in an informal survey of novice and expert computer users, everyone was able to remember situations in which their computer did something that seemed mysterious". For instance, sometimes ©Microsoft Word automatically changes "teh" into "the", but it does not change "nto" into "not" [100].

The problem of supporting users is not new. It became important in the early 1980s with the development of personal computers. Software migrated from main frame environments driven by experts, to new personal computers used by a broader public. In the context of

this migration, software had to be made understandable and easily usable by non-specialist users. In order to support users' needs, the software industry started to design so-called user-friendly interfaces and to produce manuals that would accompany their software. Such so called "user manuals" were "prepared at great **costs** by professional writers and pedagogical advisors" [28] and they aimed to "take the user by the hand to guide him/her through the sometimes painful learning process of how to appropriately use the software" [28].

However and as stated in [14], there exist inconsistencies between devices and/or software and the manuals describing how to use them. In [28] these manuals have been shown to be not enough. The author stresses that the information contained is often "very technical or not easily accessible", especially because the vocabulary is often unfamiliar to the user, and as a consequence, users need to buy third party books that explain them how to use their software. The author also states that in today's software, the emphasis on the documentation is put not so much on explaining how to use the software, but "on answering user's questions *on the fly*". But again, users have quite diverse requirements, all of which happening in various interaction contexts as expressed in [4].

Providing support "on the fly" has become the natural evolution of help systems. The aforementioned "user manuals" have continuously evolved into different forms such as *Frequently Asked Questions* or FAQs, Guides, and precomputed Tutorials, but nowadays most of support is integrated into the application, and directly accessible by users at runtime. An example of this is the *Help menu* proposed by most software, or the *Tooltips* that indicates the purpose of a button or icon at runtime.

However, this type of integrated help remains insufficient [28]. The lack of good help and support in most of the today's software is mainly due to a problem of **cost**. Software industry has become very competitive and one way to reduce the product costs is by simplifying the support that the applications provide either in the form of manuals or integrated into the user interface.

To illustrate the insufficiency of current help systems, consider for instance the car shopping website illustrated in figure 1.1. Some of the colours of the car are simply not available for

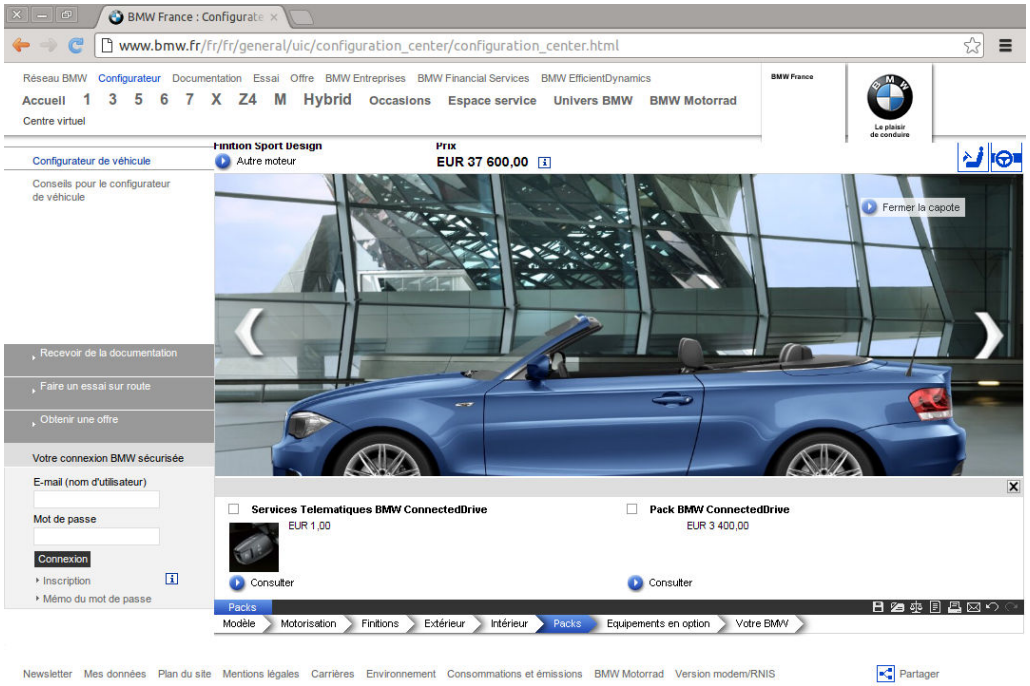


Figure 1.1: A car shopping website.

specific combinations of leathers. For example, if the user selects the “Sport Design” version of the “Cabriolet” car model, the leather colour “Boston Perlgrau” is not available. In addition, some extra equipment is simply added by default with certain car models whereas with other car models the same options can be either available or not, and it is no longer the system that chooses these options but the user instead. For instance, selecting the “Sport Design” version of the “Cabriolet” will add the Bluetooth interface for mobile phones but will suppress the “Sport Leather” option of the wheel. The bluetooth option can be added regardless the version of your model car, but the second cannot. Moreover, novice users could miss the meaning of some concepts that are used in the UI such as for instance, what does the “Finition Excellis” stand for, what is the “Shadow Line Brilliant” option, or what is the “Servotronic Direction at variable assistance”. Sometimes users simply do not know *How* to add the option they want to their current configuration such as “How to add the sport leather to the wheel?”, *Where* an option is such as “Where are the maintenance contracts?”, and other questions that

the reader can undoubtedly think of.

All the options and combinations of configurations proposed by the user interface presented in the image could be quite useful to most users, and probably they have been added to the user interface for a good reason such as because they are used by most people most of the time. However, when a novice or expert is unfamiliar with these features, “user manuals” or current help systems “on the fly” can’t simply help at that point.

All these support facilities cover most of the general topics that users may find. However, they rely on information that is written and prepared at design time. This is a limitation for the following several reasons:

- First, as static help systems rely on information that is considered at design time, these systems can’t cover all the different combinations of questions that the different users can have with regard to the user interface. For instance, in the car shopping website of the figure, a novice user could ask himself/herself “How to change the external colour of the car”, whereas an expert user could ask “How to add the Servotronic Direction to the car”. With help systems where all the information is written by hand, it becomes impossible to write all the possible questions of the users and their related answers at design time. Moreover, writing all these explanations increases the **cost** of the application, which is one of the reasons of lack of good help systems as we have previously seen.

- Second, even if the application is small enough that one can think of including all the possible previous questions by hand into the user manual, this remains an utopia. The reason is simple. Designers are not users so designers have different perceptions of the same UI than users have. In other words, as the perceptions of both designers and users are different, the potential question that they can have could be different as well.

- Thirdly, as the users’ perception is mainly based on previous experience, different users have different perceptions. Consequently, different users will potentially find different obstacles. Again, designers cannot foresee all the different problems for all the potential

users. This problem has been pointed out by many authors in the literature such as Shneiderman [136] or Myers [99].

- Finally, nowadays applications run on a diversity of platforms such desktop computers, laptops, PDAs, smartphones or tablets, presenting different user interfaces for each of them, for instance, adapting the UI to different resolutions of each platform or supporting new modalities for the interaction. This adaptation from one platform to another implies that, for the same application, options in the user interface can change, disappear, or even be modified. Tasks could be done in different ways from one platform to another, for instance, integrating gestures. Again, writing help systems that explain all the user interfaces for all the platforms is not feasible due to a problem of cost.

As applications inevitably get more and more sophisticated, help facilities will be even more necessary. The next section describes the approach followed in this thesis to deal with the previous considerations.

1.2 Thesis Approach

Many works ([80, 100, 122]) have reported on the benefits of supporting users through explanations in interactive systems. These explanations address specific questions that users ask about the User Interface (UI). For instance, *How* a task can be accomplished, *Why* a feature is not enabled, or *Where* an option is.

One approach to overcome the lack of good help without increasing the cost because of the support is to apply the model-based principles to the UI development (see for instance [61]). In this approach, the UI is directly generated from several design models that are previously created by the designers.

Based on this idea, this thesis proposes to explore the concept of Self-Explanatory User Interface as a solution to the problem of cost. A Self-Explanatory User Interface provides users with support that is automatically generated at runtime using some kind of knowledge base. This thesis explores whether the concept of Self-Explanatory UI is feasible through the design

models of the model-based approach of UIs or not, i.e., by using the models created at design time as the knowledge base at runtime, exploiting these models and the relationships between them to find answers to the users' questions.

As these help facilities rely on the same design models that are already created to construct the UI, the cost of such help facilities should be drastically reduced in comparison with other traditional solutions. Moreover, the support generated at runtime could evolve with the program specification automatically so, as the design models evolve, these help systems should automatically reflect those changes in the provided support.

Self-Explanatory User Interfaces (SEUIs) can be considered as a more concrete type of Supportive User Interfaces. A first definition of Supportive User Interfaces was published as a result of the first workshop on Supportive User Interfaces [29] in 2011. In this workshop, the participants agreed the following definition that characterizes a SUI (Supportive UI):

> A supportive user interface (SUI) exchanges information about an interactive system with the user, and/or enables its modification, with the goal of improving the effectiveness and quality of the user's interaction with that system.

According to this definition, a supportive user interface is a self-explanatory user interface that, in addition to exchanging information about the interactive system with the user, it enables its modification.

This thesis explores the state of the art on help systems, the concept of Self-Explanatory UI and its feasibility. This approach is synthesised in the working hypothesis and the thesis statement described in the next section.

1.3 Working Hypothesis and Thesis Statement

This thesis proposes a model-based approach for supporting users in the interaction process. The approach is sustained by the classical models that are used in the development of the user interface. Therefore, this fact leads us to the following hypothesis:

Hypothesis

Design models are suitable for supporting end users in the interaction process.

The immediate consequence is that design models can enrich end users' support, so they will better understand the UI. Therefore, they'll have less problems in the interaction. Therefore it is claimed that,

Thesis Statement

A model-based approach to the dynamic support of users in the interaction process, can provide benefits for the user in terms of support, increasing the quality of the interaction and the user's comprehension of the user interface.

Specific subclaims of this statement are that:

- The approach permits to provide users with different types of explanations about the user interface, for instance, "how can I do it?" or "Where is it?".
- The approach provides explicit means for requesting support (users).
- The approach provides explicit means for presenting the support back to the users (UI).
- The support provided to the users is valuable.

Based on the previous thesis statement and the subclaims, the next section introduces the research questions that will guide our research.

1.4 Research Questions

To study how to support the user in the interaction, this thesis addresses the following research questions:

Is it possible to generate explanations "for free"? The lack of good help support in most today's software is due to a problem of cost. This thesis explores whether a solution for the generation of support with a minimum cost is feasible or not.

What to explain? What type of questions self-explanatory help systems are able to answer? The explanation capabilities of a help system are restricted by the information available in the knowledge base from which the support is computed. This thesis explores whether design models are useful for supportive purposes and, if so, what information coming from these models is useful for supporting the user.

How to explain? If the user's support can be computed with information coming from one or more models, it is necessary to define a mechanism to extract this information from the different elements of each model in a first step, and then combine all these elements into a single explanation in a second step.

How to present the explanation? The computed support needs to be presented to the user in a comprehensible way. This research explores how to translate the computed support into understandable information for the user.

Is the provided support valuable? This thesis also explores if the computed support presented to users is valuable and relevant for the users, so it effectively help users to better understand the UI.

Next section details the organization of this work.

1.5 Dissertation Structure

The remainder of this dissertation is structured as follows:

Chapters 2 and 3 are related to the state of the art. The second chapter introduces the state of the art. It describes different approaches that have already contributed to supporting users in different ways. The chapter analyses these approaches and discusses their advantages and disadvantages. It then focuses on model-based solutions, identifying the possible areas of interest by defining and analysing a Problem Space.

The third chapter describes the foundations of this research. These works are necessary to understand the proposed solution. The chapter covers the different model-based initiatives first. It then explains how these initiatives have been applied to the field of HCI. It explains

the Cameleon Reference Framework along with an example, as well as the UsiXML language and some of its meta-models that are interesting for our research. Finally, the chapter ends by providing a review of the most relevant quality models of the literature.

Chapters 4, 5, and 6 describe our contributions beyond the critical analysis of the state of the art. The fourth chapter presents our conceptual contribution for building model-based self-explanatory user interfaces. The solution is based on the concept of "Gulf of Quality", an extension to Norman's theory of action. The Gulf of Quality is introduced in the chapter before presenting the design principles for building self-explanatory UIs. The chapter continues with the description of explanation strategies that are used for supporting different types of explanations. For each explanation type, an explanation strategy details how to compute the questions that the system is able to answer, and the necessary algorithms for answering such questions, illustrated with real examples and sequence diagrams.

Chapter 5 describes all the necessary elements to answer design rationale questions. The chapter starts by describing QUIMERA, a quality meta-model to improve the design rationale. It then details the relationship of the quality meta-model with the rest of the models of the system, the process for taking quality into account in the development of model-based UIs, and finally, the process of answering design rationale questions based on instances of the meta-model.

Chapter 6 presents the Implementation of the conceptual contribution. It describes a generic architecture for building self-explanatory user interfaces, the implementation details, and a running prototype. This prototype is later used in the evaluation of the approach, which is also described in the chapter. The chapter ends with a discussion about the findings and observations issued from such evaluation.

Chapter 7 discusses the conclusions and future work.

Figure 1.2 provides a visual structure of the thesis organization, showing how the different contributions are distributed through the chapters.

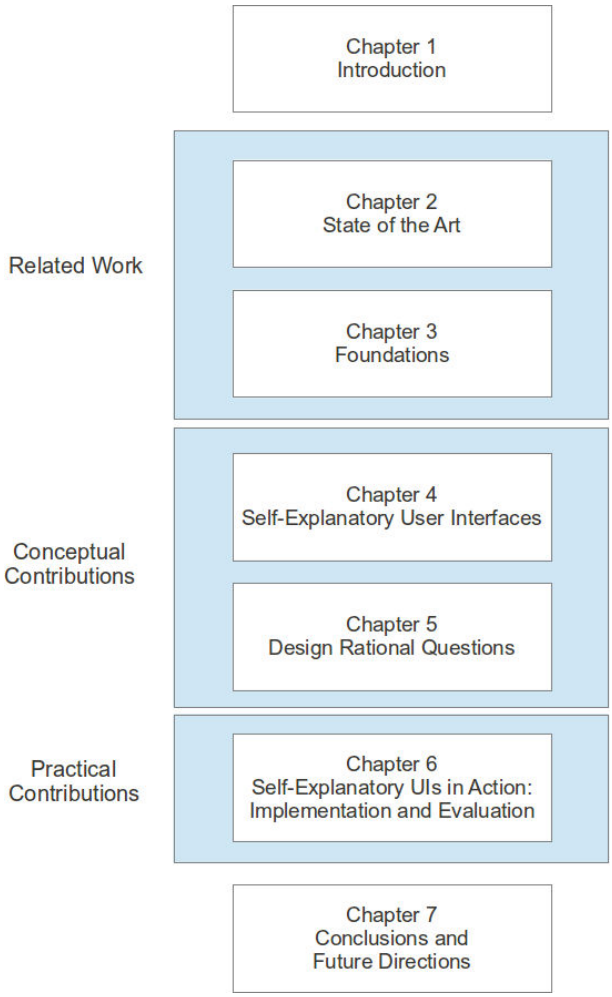


Figure 1.2: Thesis structure.

2

State of the Art

> *If I have seen further, it is by standing on the shoulders of giants.*
>
> Isaac Newton,

This chapter reviews previous works from several computer science fields that provide explanations to better support users during their interaction with the system. We begin by giving a detailed overview of the concept of explanation. We then review the most relevant works about explanation from several research domains, including knowledge-based systems, intelligent agents, recommender systems, as well as other help systems that cannot be categorized into these approaches. We also review the different explanation taxonomies developed in several of these research domains.

After reviewing these works, we focus on model-based solutions. We compare different model-based propositions of some reviewed authors through a *Problem Space* subdivided in different areas and axes. This problem space has helped us to identify areas of interest for our research.

We finally discuss how we draw inspiration from these model-based works that have investigated explanations over the past several decades, identifying gaps and opportunities for providing explanations through a model-based approach of user interfaces.

2.1 What is an Explanation

The concept of explanation has been addressed by many philosophers, scientists and researches along the history. This section gives an overview on how the term explanation has evolved through different explanation theories, from narrow definitions covering only the relationship of causality, to broader interpretations addressing a larger number of concepts. This overview sets the basis to understand what are the current dimensions covered by the term explanation nowadays, and which of them are meaningful in computer science and specially in the context of this research. These dimensions are presented at the end of the section.

2.1.1 Theory of Explanation in Philosophy of Science

In the Philosophy of Science, the main kind of explanation are scientific explanations. Aristotle's Theory of Causality is considered as one of the ancients theories of explanation. This theory explains an event or a phenomenon by identifying its cause. In other words, the explanation of why something did happen is an event or phenomenon inducing the causation. Scientific explanations traditionally followed this definition, trying to explain some facts in terms of some laws. For instance, one classical definition of explanation in these terms is given in [129] as follows:

> Can some fact E (the explanandum) be derived from other facts A thanks to the application of general laws L (the explanans $L \cup A$)?

The definition of explanation remained related to the concept of causation until the 20th century. In this century, the concept of explanation began to evolve through different theories of scientific explanation[1]. In this period, explanations are treated either in a realist sense -the explanation is a literal description of the external reality[2]- or in an epistemic (anti-realist)

[1] For a detailed description of the development of theories of scientific explanation since Hempel's earliest models in the 1940's, see[129]

[2] Descriptions according to `http://www.iep.utm.edu/explanat/`

sense -the point of an explanation is only to facilitate the construction of a consistent empirical model, not to furnish a literal description of reality[2]. This epistemic approach was the starting point for Hempel to develop the epistemic Theory of Explanation [57] in 1948. In the epistemic Theory of Explanation, explanations are exclusively based on a logical approach. This theory had an important impact in the evolution of the concept of explanation and the understanding of (scientific) explanations. Based on the Hempel's Theory of Explanation, Prior and Prior proposed the Erotetic Logic [121] in 1955 for the analysis of questions using a formal logical approach.

2.1.2 Erotetic Logic: Subject and Request

Erotetics [121] is the part of the logic devoted to the logical analysis of questions. Its formal logical approach decomposes questions into two parts: the *subject* and the *request*. The *subject* does not refer to the grammatical subject of the question but "the possible states of the world that are presupposed by the question" ([119]). The *request* identifies "how many of these states are desired in the answer" ([119]) . For instance, in the question "Is there a model of this car having a diesel engine?", the set of possible alternatives is that there is such a model or there is not. This set of alternatives forms the subject. The request identifies that the desired answer is one that specifies which of the alternatives states is true: that either there is or there is not a car model with a diesel engine.

Erotetic Logic opens a new perspective on explanations from the point of view of questions, based on the *subject* and the *request*. Other approaches decompose questions into subjects and requests in the same way the Erotetic Logic does. The Jahoda and Braunagel's approach in 1980 uses a similar decomposition but in terms of *given* and *wanted* elements. As stated in [119],

> According to Jahoda and Braunagel, the given is *the subject of the information need*, and the wanted is *the type of information needed about the subject*.

For example, in the question "I am looking for this car model with a diesel engine" the

given is the diesel engine and the *wanted* is the car model.

For the scope of this research, we take from Erotetic Logic the idea that questions may be decomposed into a subject and a request.

Different Theories of Explanation provided different insights on the concept of explanation. It is the case of the term *structural explanation* described next.

2.1.3 Structural Explanations

Structural Explanations were firstly introduced as a kind of scientific explanation (see for instance [32]). We talk about a structural explanation when the properties or behaviour of a complex entity are explained by alluding to the structure of that entity [92]. A non-formal definition of structural explanations is given by Hughes in [60]:

> A structural explanation displays the elements of the models the theory uses and shows how they fit together. More picturesquely, it disassembles the black box, shows the working parts, and puts it together again. "Brute facts" about the theory are explained by showing their connections with other facts, possibly less brutish.

Our research takes from the concept of *structural explanation* the idea of supporting the user of the User Interface by composing explanations with the relevant elements behind this User Interface. In our particular case, i.e., the model-based approach of user interfaces, these elements are the underlying models of the User Interface from which this User Interface is generated. In terms of Hughes, these underlying models and their different elements are the pieces of our "black box", i.e., the UI itself.

If *Structural Explanations* are a type of explanation that employs the modules, parts, or sections of an entity to answer specific questions about that entity, many other different explanation types started to be developed with the advancement of the computer science in the last quarter of the twentieth century. The next section reviews the most relevant explanation types addressed by different computer science approaches in a chronological order.

2.2 Approaches

During the development of the computer science in the twentieth century, different computer science domains addressed the problem of supporting users in the interaction with the systems using some forms of explanations. This section summarizes the most relevant contributions for each of these domains (expert systems, agents, explanation facilities, etc.). The presentation of each approach starts with an *introduction* explaining the relevant concepts and terms that are necessary to understand it, and the related work illustrated with some examples.

2.2.1 Expert Systems

According to [105], expert systems are considered as "the first truly successful forms of Artificial Intelligence software". They were introduced by Edward Feigenbaum in the 1970s with the Dendral system, and actively developed in the 1980s [76].

2.2.1.1 General Principles

In artificial intelligence expert systems are devoted to, among other objectives, explain and guide the user during the interaction process with the system. According to [63], an expert system is defined as

> "A computer system that emulates the decision-making ability of a human expert."

In order to be considered useful and acceptable, expert systems must be "able to explain their knowledge of the domain and the reasoning processes they employ to produce results and recommendations" [97].

Researchers have identified different reasons why the explanation capabilities of the expert systems are "not only desirable, but necessary" [97]. Some of these reasons include [16]:

1. Assisting both users and system builders in understanding the contents of the system's knowledge base and reasoning processes.

2. Facilitating the debugging of the system during the development stages.
3. Educating users both about the domain and the capabilities of the system.
4. Persuading users that the system's conclusions are correct so that they can ultimately accept these conclusions and trust the system's reasoning powers.

Expert systems were firstly structured into two well distinguished parts: the *inference engine*, and the *knowledge base*. The inference engine is fixed and independent from the expert system. The knowledge base is variable, and is used by the inference engine to perform the reasoning. This division originated the sub-family of expert systems called *Knowledge-Base Systems (KBS)*.

2.2.1.2 Knowledge-Based Systems

Knowledge-Based Systems (KBS, also known as *Rule-Based Systems*) focus on the underlying information -or base of knowledge- represented or modelled inside the system itself. Among the most popular KBSs of the 80's are XPLAIN [142], NEOMYCIN [25] and EMYCIN [151]. According to Gregor and Benbasat [54], KBSs cover the following four different types of questions or categories:

What is - This type of questions provide information about specific terms or domain concepts. This category was identified as *Terminological* by Gregor and Benbasat and used by Swartout and Smoliar in [144].

Why (System logic) - Gregor and Benbasat identified this category as *Control or strategic*. Answers to these questions provide explanations about the "system's control behavior, and problem solving strategy", giving an insight into the design rationale of the system logic. This kind of explanations are used in expert systems of this period such as the NEOMYCIN system [25].

Why (Reasoning) - These questions (also called *Trace* or *Line of reasoning* as in [54]), explain the processes taken by the system to come up with its results. The explanations belonging to this type of questions were used as well by several experts of this time such as the EMYCIN [151] system.

Why (Justification) - The *Justification* category as named in [54], was used by expert systems such as XPLAIN [142]. They provide the so called "deep explanations" about design rationale justifications. For instance, in XPLAIN these explanations were used to provide justifications of the code, not explaining what the code does but its rationale.

Some of these systems started to use different types of models as their knowledge base. For instance, in XPLAIN the author states that the system "uses a domain model, consisting of descriptive facts about the application domain, and a set of domain principles which prescribe behavior and drive the refinement process forward".

These models are here in the form of rules. Using models in any form is a recurrent solution used by other explanation systems and not only by the expert systems. Our research takes from expert systems the idea of adopting the system knowledge (in the form of models) as the source of knowledge that is used to extract the necessary information for the users. In the context of this research, these models are the same that are used to build the user interface.

In the later 70's and the beginning of the 80's, expert systems were mostly presented in a command-line form. Over the 1980s, researchers added a third component to this structure, namely a *dialogue interface*. The role of the dialogue interface is to conduct a conversation with the users. These interfaces were later called "conversational interfaces" and were one of the starting points for what are now called *intelligent agents*.

2.2.1.3 Intelligent Agents and Cooperative Support

Software agents, or simply called agents, appeared in the early 1990s. According to [9], software agents are defined as:

> Entities capable of voluntary, rational action carried out in order to achieve goals and holding a representation or 'belief' in the state of the world. They come to hold these beliefs through existing data and by deriving new belief from interaction with external sources and as a result of internal reasoning.

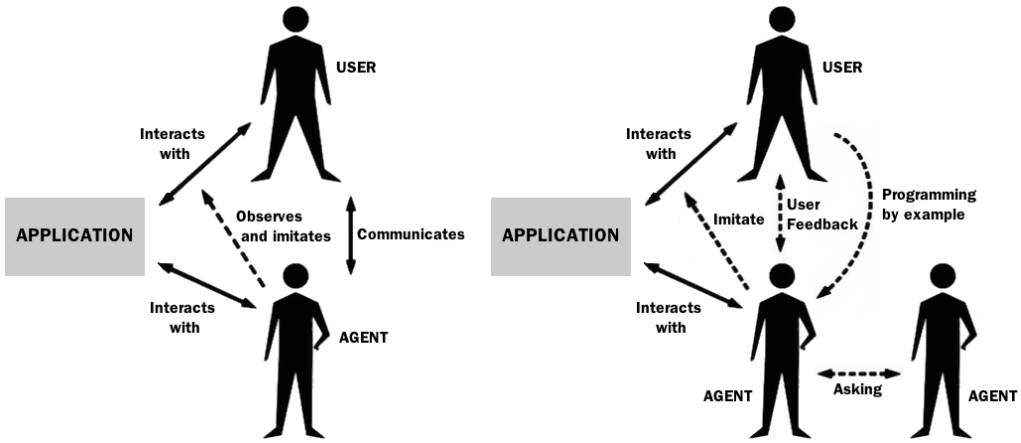


Figure 2.1: Different roles of agents. Adapted images from [85].

Other authors propose similar definitions (Bradshaw [15] in 1997, Weiss in 1999, Russell and Norvig in 1995, or Hayes-Roth in 1995).

Agents try to support users with some tasks. Users can delegate responsibilities to these agents, so the agents will perform the necessary actions to accomplish the expected tasks. Agents are usually able to learn from single users or even from other agents thanks to the different learning facilities that they integrate. These facilities are based on behavioural patterns, that permit to observe and imitate the actions that the user performs on the user interface[3]. Figure 2.1 adapted from [85], summarizes the role of software agents. On the left side of the image, the interface agent does not act as an interface or layer between the user and the application. Instead, it behaves as a personal assistant that cooperates with the user on the task. The user is able to bypass the agent. On the right side of the figure, the interface agent learns in four different ways:

1. it observes and imitates the user behaviour
2. it adapts its behaviour based on user feedback
3. it can be trained by the user on the basis of examples
4. it can ask for advice from other agents assisting other users

[3]Interested readers can find a review of agent-based interaction, implementations and design guidelines in [87] and [85].

Agents have been classically devoted to assist users in the interaction with the system, trying to perform routinary tasks in order to improve the user experience. For instance, they can perform repetitive tasks, which has been proved to increase the users' efficiency while performing tasks.

Agents are proactive by definition. They propose help to users when they consider that it is necessary. One of the most popular agents, in part due to its continuous suggestions, is *Clippy*, the office assistant of the Microsoft Office(©) suite.

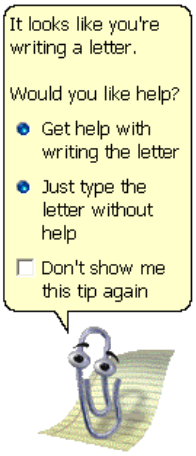


Figure 2.2: Clippy.

Clippy was able to assist users by means of an interactive animated character (figure 2.2), which interfaced with the Office help content. Clippy proposed different type of content that it estimated helpful for the user according to the user actions. Clippy is not a collaborative agent, this is, Clippy does not communicate with other agents as shown in the right side of the figure 2.1. For frameworks showing multi-agent collaboration, please see [52, 71, 27, 145].

Collaboration between agents for supporting other agents is done with several techniques, always based on some kind of language or protocol. Examples of these are the knowledge interchange languages and agent communication languages, such as KQML. KQML [39] enables human-agent and agent-agent communication and co-ordination. As stated in [4], agents can be coupled as well with "advanced interface components (e.g. speech, facial animation, etc), which make anthropomorphic agent emulation feasible". An analysis of agents can be found in [134], and a more detailed discussion of agent-based systems in [65].

According to [55], agents provide explanations according to four different explanation types[4]: ***what***, ***how do I***, ***how does it work***, and ***why*** (design rationale). These four groups of explanations are also referred in the literature as *ontological*, *operational*, *mechanical*, and *design rationale* [55]. The explanations given by an agent covers sometimes information about

[4]For a complete review of explanations in intelligent agents see [55].

the agents themselves. For instance, the group of *what* explanations provides information not only about definitions or terminology about the domain of the system, but also about the identity of the agent, or the relations between agents or components of the system.

The explanations behind the Design Rationale category stated by Haynes in [55] addresses *why* questions that cover different aspects of the design rationale of a system. Haynes categorizes these aspects into four different classes: *Deductive-Nomological* explanations using laws to describe relations between system components or agents, *Functional* explanations that provide information about the purpose of a component or agent, *Structural* explanations describing the structure of the system constraints that cause an entity or event to happen, and *Pragmatic* explanations that provide answers to questions such as what if or why not.

In parallel to the evolution of expert systems in its different forms such as KBSs or intelligent agents, question-answering systems helped to the development of help systems by providing several explanation taxonomies in the form of questions and answers. The next section covers these taxonomies.

2.2.2 Question Answering Systems

This section describes what Question Answering systems are, the type of questions that they address, and the main taxonomies of questions types that have been developed for such systems. Some of these classifications of question types have inspired ulterior question types for help systems.

2.2.2.1 General Principles

Question-answering (QA) is defined[5] as

> A computer science discipline within the fields of information retrieval and natural language processing (NLP), which is concerned with building systems that automatically answer questions posed by humans in a natural language.

[5]Description according to `http://en.wikipedia.org/wiki/Question_answering`

The first QA systems were basically natural language interfaces for expert systems focused on specific domains. In contrast, the question-answering systems available nowadays use text documents as its knowledge base and combine various techniques of natural language processing.

QA systems are classified according to the nature of the domain of the questions:

Closed-domain QA Systems dealing with questions under a specific domain. They normally exploit domain-specific knowledge frequently formalized in ontologies. Most of QA systems in this category answer only a limited type of questions.

Open-domain QA Systems dealing with questions of almost any type about nearly anything.

Interested readers can refer to [6] for a more detailed review of the different QA approaches.

The main goal of QA systems is not to help users in the interaction with a specific system, but our research pays special attention to this discipline because QA systems have greatly contributed to classify the different possible types of questions that a system needs to deal with. Different authors (see for instance [75]) have proposed several question taxonomies in the last decades that have been lately used not only by QA systems but also by help systems from other disciplines such as expert systems or many help facilities. The next section reviews the most relevant classifications of questions.

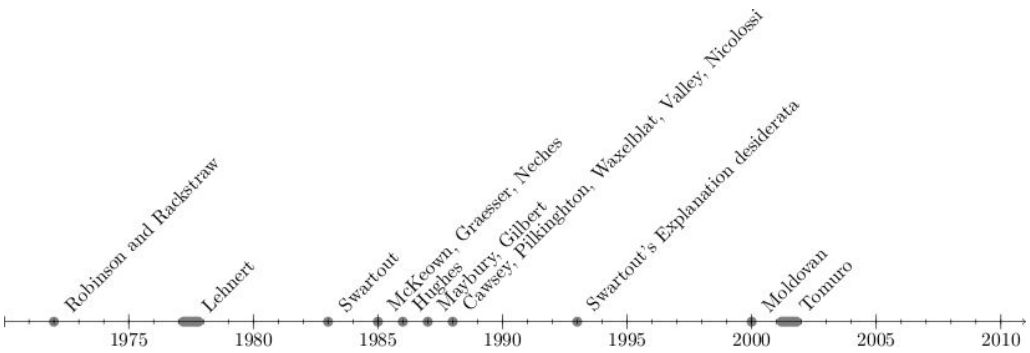


Figure 2.3: Evolution of question classifications by authors in the last forty years. After 1990, most of the classifications reuse the same question types.

2.2.2.2 Question Types in Question Answering Systems

Figure 2.3 shows the evolution of some question classifications of the last years in QA systems. The figure shows how most of the work was done before the nineties and, in fact, current research involving question types are mostly based on classifications of question types created at this period. This section only discusses those that have been found to be relevant for our research.

In 1972, Robinson and Rackstraw [125, 126] proposed a lexical classification of questions based on the words that can be used to form an interrogative sentence. They proposed the following seven categories:

1. Who
2. Which
3. What
4. When
5. Where
6. Why
7. How

This classification based on question types is simple and it has been used by several Question Answering systems [58, 96]. One of the problems of this classification is that, for open questions, some of these questions can be wrong classified regarding only the lexical perspective. For instance, the authors consider that *what* questions are expected to be answered with definitions. However, the expected answer of the question ***What*** *I need to do to configure a car with a diesel engine?* is related to the process of configuring itself which is covered by questions of type *How* instead. To overcome this limitation, help systems relying on this set of questions or other similar classifications, tend to propose to users a closed set of questions in which users are not able to ask open questions but, on the contrary, select a question among those proposed by the system.

Other contemporary authors also focused in particular *Wh*- question types. For instance,

Swartout concentrated in 1983 on two different question types in his expert systems [143, 101]:

1. Why
2. How

These two questions are still widely used in different help systems nowadays and not only in the expert systems domain.

In 1985, McKeown proposed a question classification [91] in her *Text* system, based on a natural-language database. McKeown argued that several experiments [146, 108] "*shown that users often need to ask questions about database structure to familiarize themselves with it before making requests about its contents*". For this reason, McKeown proposed three question types corresponding to three communicative goals:

1. Request for definitions
2. Request for available information
3. Request about the difference between database entities

What is important in McKeown's requests is the inclusion of a general category named *Requests for available information*. We consider it as well as an interesting dimension of explanations for help systems in HCI in terms of *Availability* of actions, because according to the studies cited by McKeown [146, 108], this could be potentially useful for novice users of a system.

In the same year, Graesser and Murachver 1985 proposed a classification of question types in which seven types are cross-classified with three "statement categories", forming a total of 21 categories in all (see table 2.1). The authors can then characterize a question as a function of the form:

QType(QConcept, Knowledge)

QType is the question type, for instance *What, Who, Why, ...* *QConcept* refers to what the question is 'about', classified along three statement categories: *Action*, *Event* and *State*.

The *Knowledge* attribute is the knowledge base used to answer the question, for instance, a natural-language database. Table 2.1 summarizes this classification.

Question Types	Question Concepts
- Why - How - Enable - Cons (what is the consequence of) - When - Where - Sig (what is the significance of)	- Action: A behaviour by an animate actor which is directed toward a goal - Event: A change of state in the physical or social worlds or in the mind of an animate being - State: An ongoing characteristic of an entity or a relationship between entities

Table 2.1: Graesser and Murachver's question classification [53]

What is interesting in this approach is the explicit relationship between question types and answers, as well the practical approximation to the erotetic logic in which subject and requests are explicitly represented. In our research, we keep the idea of explicitly linking question types and answers as in the *QType* relationship.

In 1988, Cawsey defined [20] the question types along three orthogonal dimensions. The first dimension refers to the nature of the inquiry, regarding whether the subject was requesting information, suggesting information, or confirming information with the system. The second is the type of the question. The third dimension is the type of information involved in the question. Table 2.2 summarizes the three axes with their respective categories.

Cawsey classifies question types according to their syntax. The dimension named *Type of information* does not include some type of questions that can be useful for supportive purposes as, for instance, procedural questions. Cawsey proposes however a category called *rest* in which all the questions that do not fit any other previous category can be classified. The same principle applies to the *Type of question* category where unconsidered types of questions fall in the *Others* category.

In 1988, Waxelblat proposes an alternative question classification in [155]:

1. How shall I do what you ask me to do?
2. Why do you ask me to do that?

Category	Type of Information
Whether the subject was:	- requesting information - suggesting information and asking for confirmation - repeat/rephrase information to check if it is understood
What type of question is asked	- What - Why - Others (What if, Why not, How)
What type of information was involved	- Behaviour - Part recognition - Function - Notation - Structure - External actions - Limits and Assumptions - Input, output and variables - Concept - Information - Rest (uncategorised)

Table 2.2: Cawsey's question classification

3. How did you come to that question or conclusion?
4. By what steps did you get here?
5. What shall I do next?
6. What do you know about?

Waxelblat's classification relies on a rigid and small set of questions. However, we can see the interest of the author in questions about the behaviour of the system (question 3), or about the procedures (questions 1, 4 and 5). As in Waxelblats' classification, these particular question types have been repeatedly proposed by different authors due to their interest for the users. For this reason, in this research we keep the same interest about behavioural and procedural questions.

In parallel to the development of classifications of question types based on *Wh*- words, the QA community started to classify questions and answers according to the function of the question. In 1977 and 1978, Lehnert proposed [74, 75] one of the most used questions classification for open-domain QA systems. Lehnert defined thirteen types of questions (see table

2.3). Not all of them are directly useful for help systems because these types were specifically designed for open-domain QA systems, but authors from expert systems as well as authors from closed-domain QA systems started to focus on some particular Lehnert's types. As an example, interested readers can refer to the Pilkinghton's question classification [118], the Nicolosi's question classification [103], or the Valley's question taxonomy [150], all of them proposed around 1988.

As readers can notice, most of the explanation taxonomies either in the form of question types based on *Wh*- words, answer functions, or a mix of them, were developed more than fifteen years ago. Recent question classifications in QA systems are mostly based on these classifications, adding specific types for concrete domains. For illustration, a more recent classification proposed by Tomuro [148] in 2001 and reviewed in 2002 proposes new categories regarding the function of the answer, as a result of an automatic extraction of question terminology from a corpus of questions classified by question type. The result of this open-domain question type classification is summarized as follows:

1. Definition
2. Reference
3. Time
4. Location
5. Entity
6. Reason
7. Procedure
8. Manner
9. Degree
10. Atrans
11. Interval
12. Yes-no

As a synthesis about question types and question classifications, we can conclude that there are two main different question classifications, one based on *Wh*- words mainly used by expert systems, and a second one relying on the function of the question/answer, initiated by Lehnert in [74] and extensively reused for open-domain QA systems. The common points of both approaches are that first, they rely on a set of limited categories. Second, they both identify questions by its nature or type, either in their lexical form in the case of *Wh*- classifications, or according to their function as in [74]. Third, in both classification systems there

Question Categories	**Examples**
1. Causal Antecedent	Why did John go to New York? What resulted in John's leaving? How did the glass break?
2. Goal Orientation	How did the glass break? For what purposes did John take the book? Why did Mary drop the book? Mary left for what reason?
3. Enablement	How was John able to eat? What did John need to do in order to leave?
4. Causal Consequent	What happened when John left? What if I don't leave? What did John do after Mary left?
5. Verification	Did John leave? Did John anything to keep Mary from leaving? Does John think that Mary left?
6. Disjunctive	Was John or Mary here? Is John coming or going?
7. Instrumental/Procedural	How did John go to New York? What did John use to eat? How do I get to your house?
8. Concept Completion	What did John eat? Who gave Mary the book? When did John leave Paris?
9. Expectational	Why didn't John go to New York? Why isn't John eating?
10. Judgmental	What should John do to keep Mary from leaving? What should John do now?
11. Quantification	How many people are there? How ill was John? How many dogs does John have?
12. Feature Specification	What color are John's eyes? What breed of dog is Rover? How much does that rug cost?
13. Request	Would you pass the salt? Can you get me my coat? Will you take out the garbage?

Table 2.3: Lehnert's 13 conceptual question categories.

Question Types	Examples
Goal-Oriented	What can I do with this program?
Descriptive	What is this? What does it do?
Procedural	How do I do this?
Interpretive	Why did this happen?
Navigational	Where am I? Where is it?

Table 2.4: Sellen et al. question classification

are question types that appear repeatedly because of their relevance, as for instance *Why* or *Behavioural* questions, and *How* or *Procedural* questions.

In order to answer some of these specific types of questions, the next section describes some model-based works that propose model-based explanations.

2.2.3 Model-based explanations

The Model-Based approach can be defined as:

> A software development paradigm that focuses on the creation and exploitation of domain models, i.e., abstract representations of knowledge and activities that govern a particular application domain.

The first tools to support model-based explanations were the Computer-Aided Software Engineering (CASE) tools developed in the 1980s. These tools evolved in parallel with expert systems and it is normal to find in the literature influences from one discipline into the other, typically in the types of questions supported by each approach. This section briefly explains how different types of models have been used in different works for supporting users with different types of explanations that vary regarding the nature of the question asked by the user. For instance, [132] described five categories of questions (see table 2.4).

Other authors describe similar categories but with different terms. For instance, in [139] we find:

1. Conceptual explanations (What is this?, What is the meaning of this?),

2. Why-explanations describing causes and justifications for facts,
3. How-explanations for describing processes,
4. Purpose-explanations (What is this for? or What is the purpose of this?),
5. Cognitive explanations, which "*explain or predict the behaviour of 'intelligent systems' on the basis of known goals, beliefs, constraints, and rationality assumptions*" (adapted from [139]).

As we can see in this classification, we retrieve the same similar question types from most of the classifications of the previous section.

Many works report on the use of different models for specific explanation types. The rest of the sub-sections reviews some of these works categorized by the types of models they rely on.

2.2.3.1 Task Models

The Task Model is probably one of the most used models for providing explanations. An early example that employs a task model (in the form of user's actions) for explanation purposes is Cartoonist [141]. Cartoonist generates GUI animated tutorials to show a user *How* to accomplish a task, exploiting the model for providing run-time guidance.

Pangoli and Paterno [114] allow users to ask questions such as *How can I perform this task?* or *What tasks can I perform now?* by exploiting a task model described in CTT notation [115]. Contrary to Cartoonist, answers are provided in [114] in a pseudo-natural language form.

Tasks modelled in the form of Petri nets [116] are used for similar purposes by Palanque et al. in [113], answering questions such as *What can I do now?* or *How can I make that action available again?*

Other works report on the usage of task models as a means for creating collaborative agents able to helps users. An example can be found in [35].

Task models have also been used for supporting purposes in the so called task processing systems. McGuinness et al. [90] identified several explanation types in the context of task processing systems. These explanation types are summarized in table 2.5.

Question Types	Examples
Motivation for tasks	why are you doing a task?
Task status	what task is being done? what the status of the task is?
Task history	what the system has done recently what it has started recently why it did a task (in the past, as opposed to why it is doing) why it didn't do a task how it did a task
Task plans	what the system will do next when it will start the task and why how it expects to do it
Task ordering	why a task is being done before another why some task has not yet started what needs to be done to complete a task
Explicit time questions	when a task will begin or end how long a task took to complete why a task took so long to complete why a task is already being done instead of later

Table 2.5: Mcguinness et al. question classification for task processing systems.

In some of these task processing question classifications as in the previous one, we find behavioural explanations answered with task models. Next section covers other models used by different works for the purpose of explaining the behaviour of a system.

2.2.3.2 Behaviour Models

Behaviour models, presented in different forms, have been also used to support users through questions such as *Why* or *Why not*. In [9] *Why* questions are answered using the same approach based on Petri nets that is exploited for procedural questions. By analysing the net it is possible to answer questions such as *Why is this interaction not available?*

The Crystal application framework proposed by Myers et al. [100] uses a "Command Object model" that provides developers with an architecture and a set of interaction techniques for answering Why and Why not questions in UIs. Crystal improves users' understanding of the UI and help them in determining how to fix unwanted behaviour.

Vermeulen et al. [153] propose a behaviour model based on the Event-Condition-Action

(ECA) paradigm [3], extending it with inverse actions (ECAA-1) for asking and answering why and why not questions in pervasive computing environments.

Lim et al. [78, 80] observed that why and why not questions improve users' understanding and confidence of context-aware systems.

As we can see, *Why* questions are answered by different authors using different model-based approaches. The next section describes a different type of help systems that rely on the Internet instead of models for supporting users at runtime.

2.2.4 Social-Network Based Systems

Some help facilities make a strong focus on how to exploit Internet as their external source of knowledge, taking advantage from the interconnection of users. This is the case of the system proposed in 2012 by Jeffrey Nichols and Jeon-Hyung Kang in [102]. This system uses social networks as its knowledge-base.

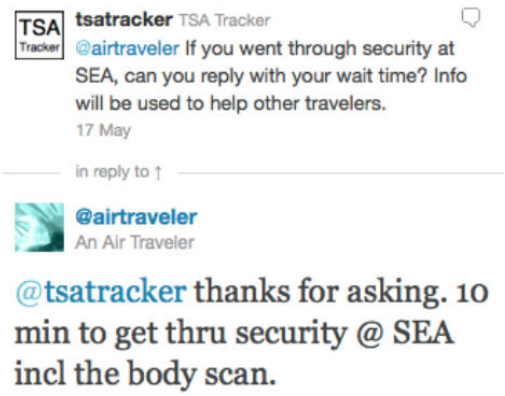


Figure 2.4: An example question/answer regarding the airport security wait time at SeaTac Airport (image from [102].

The system asks questions of targeted strangers on social networks as *Twitter*[6]. An example of the system is shown in figure 2.4. The authors state that when people have questions,

[6]Twitter©(`http://twitter.com/`) is an online social networking service and microblogging service that enables its users to send and read text-based messages of up to 140 characters, known as *"tweets"*

they often turn to their social network for answers. If the answer is obscure or time sensitive however, no members of their social networks may know the answer. This work explores the feasibility of answering questions by asking strangers on social networks. The questions supported by this work can be, potentially, of any possible type. However, questions cannot be always answered if strangers do not know the answer, and the approach requires a permanent connection to the Internet to work.

2.2.5 Personal assistants

Internet is also the source of knowledge employed by different personal assistants. Personal assistants have emerged thanks to the development of mobile devices. An example of a personal assistant is Siri. Siri is described[7] as "an intelligent personal assistant and knowledge navigator that uses a natural language user interface to answer questions, make recommendations, and perform actions by delegating requests to a set of Web services."

Figure 2.5: Siri in action.

To use the Web services, Siri (figure 2.5) also needs an Internet connection as well. For instance, Siri relies on *Bing Answers*, the *Wolfram Alpha* engine, and *Evi* for factual question answering. Moreover, Siri works only for a very limited specific set of platforms (those from Apple). Siri recommendations are similar to those provided by recommender systems, that are explained in the next section.

[7] Definition according to `http://en.wikipedia.org/wiki/Siri_(software)`

2.2.6 Recommender Systems

Recommender Systems are defined [124] as

> A subclass of information filtering system that seek to predict the 'rating' or 'preference' that a user would give to an item (such as music, books, or movies) or social element (e.g. people or groups) they had not yet considered, using a model built from the characteristics of an item (content-based approaches) or the user's social environment (collaborative filtering approaches).

Recommender systems usually rely on data gathered by user profiles. Most of the explanation types used by recommender systems are founded on similarities of the attributes of the products or entities of the system. Tintarev [147] classifies the explanations used in recommender systems in several types such as case-based, content-based, collaborative, demographic, and knowledge-based. Some examples of such systems are the *Amazon* recommender system that will recommend additional items based on a matrix of what other shoppers bought along with the currently selected item, the *Netflix* recommender system that offers predictions of movies that a user might like to watch based on the user's previous ratings and watching habits and the characteristics of the film, and the *Pandora Radio* recommender system, that "takes an initial input of a song or musician and plays music with similar characteristics" based on a series of keywords attributed to the selected artist or piece of music). The stations created by *Pandora* can be refined through user feedback (emphasizing or deemphasizing certain characteristics).

2.2.7 Desktop facilities

Other works have made focus on providing a concrete solution for a specific type of question. This is the case of the Apple location facility (figure 2.6). Apple uses it on the Help menu of every application on their desktop platform in order to help users with the navigation. In the example shown on the left side of the figure, different icons are highlighted when the user

types keywords in a searchbox. The figure on the right side shows how an arrow indicates where the required menu option is located when the user types relevant words in the same searchbox. This facility is limited in the sense that it only covers *Navigational* or *Where* questions but, on the contrary, its answers are very clear because they use the real options from the user interface that the user is asking for.

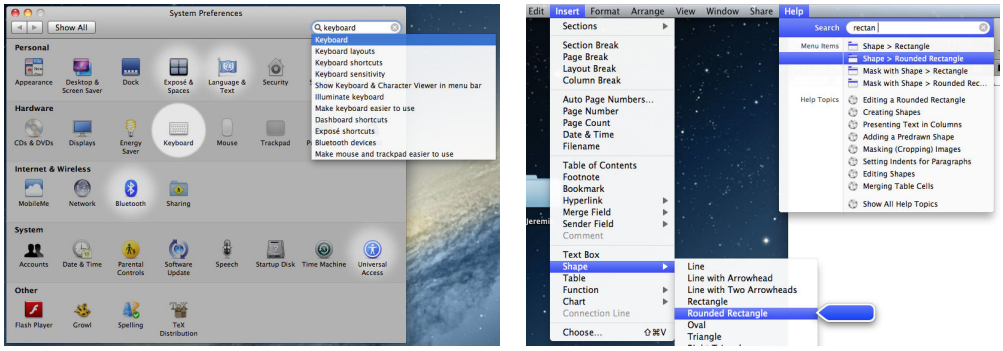


Figure 2.6: Apple location facility.

2.2.8 Avatars

As well as recommender systems give explanations in response to a specific domain (most of the time for selling or configuring products), other explanation facilities provide explanations about the context, specially in critical context where users may need assistance. It is the case of the avatar shown in figure 2.7. The avatars of the figure are located at the entrance of the Birmingham airport. The avatar tries to assist the travellers giving some directions about the airport, the customs (airport luggage-check area), and others. The user interface is not interactive at all and the video(s) plays in a loop. Even if the avatar do not have any form of "sensing the context" -for instance, they were not equipped with any kind of sensors to detect the presence of a traveller in front of the avatar-, it clearly shows an effort to gain the confidence of the users by representing a real human-being.

Figure 2.7: Avatars providing multimodal information at the Birmingham Airport (picture taken in September 2012)

2.3 Analysis of the approaches

This section contrasts the related work covered in previous sections. To this end, we used different criteria which is presented first. For each criterion, a discussion about how it applies to the different approaches is provided. After comparing these works and approaches according to these criteria, the section finally discusses the possible benefits of a model-based approach for explanations compared to previous solutions, arguing why it is worth to explore this approach for help systems.

2.3.1 Criteria and their application

This section describes different criteria that helps to identify advantages and disadvantages of each of the presented approaches.

2.3.1.1 Coverage of Questions

The number and types of questions that a help system is able to answer can be used as a criterion for comparing such help systems.

Some approaches are centred on one unique type of question. This is the case of some desktop facilities such the Apple location facility presented in section 2.2.7, that has been de-

signed for answering exclusively *Where?* questions. In the same manner, recommender systems provide information about differences (*What are the differences?*). Also, avatar systems as the one described in section 2.2.8 are intended to provide only a reduced amount of information (for instance, information about exits of the airport and the customs). Moreover, the avatar does not let the users to choose the information that he/she wants to know, because the information messages are displayed in a sequential way. Personal assistants can however answer a wider set of questions as they usually rely on web-services such as those that we have described for *Siri* in section 2.2.5.

With respect to the rest of the approaches, the number and type of questions that these approaches can cover is significantly bigger. Expert systems cover different questions, question answering systems can even support open questions, and we have also reviewed different works based on model-based approaches that provide explanations for different questions such as Procedural questions as *How* or Behavioural questions as *Why*.

Of course, those systems that allow for open-domain questions such the QA systems, some personal assistants, and social-network based systems, can hypothetically cover any type of question. The problem with those systems is not the number nor type of questions, but the quality of their answers, as explained in the next section.

2.3.1.2 Quality of Answers

The quality of an answer is subjective to each user. However, we can consider some factors that can make a difference between different answers:

1. Availability of the answer
2. Time of response
3. Reliability
4. Security

Some works constrain the way in which the user can obtain the answer. For instance, in some of the previous works such as social-network based systems or personal assistants,

the help system needs to be connected to the Internet to be able to provide an answer. This is what **availability** means. Social-network based systems and Personal systems relying on web-services require a permanent Internet connection to support the user. This can be a serious limitation regarding the context of the user. For instance, users cannot be assisted by these technologies in a hospital. On the contrary, expert systems, QA systems and model-based systems, can continuously support the users as they are usually embedded into the application.

As part of the quality in the answer, the **Time of response** can be discriminant for critical situations requiring fast support. Here again, all the previous approaches that rely on the Internet or those that need to wait for other users to answer such in social-network systems, are for these reasons slower than implementations of classical approaches such as expert systems or model-based systems where the help is usually embedded into the application. In the case of the avatars, as the users can't choose the desired information (all the help is provided in a cyclical way), the time of response can be critical if users cannot wait to access the information (for instance, for the presented avatar in the airport, because the passengers/users need to go aboard).

The **Reliability** of the answer or explanation provided by a help system is dependent of the source of knowledge used to retrieve the explanation. All the systems relying on external services, specially those that rely on external users like the social-network based systems, could provide answers that are not reliable if the user providing the answer has not enough knowledge about the question. Classical approaches like expert systems or model-based systems require however to capture the knowledge necessary for answering the question in some form, for instance in a knowledge base in the case of expert systems, or in models for model-based systems, so users can find correct answers to their questions. This information usually comes from experts of the domain with a deep understanding of the application.

In conclusion, reliability answers usually requires that the necessary knowledge about the application is available to the help system.

Security of help systems is another factor that can affect the users' experience. This is crucial in social-network based systems where users cannot openly ask about critical aspects for them, such as privacy related questions. For instance, in a user authentication banking system as the one shown in figure 2.8, a user cannot confirm if he/she has correctly introduced his/her personal identification number or PIN by using help systems relying on unknown external users as in the social-network based systems, because the user must not share private information such as his/her PIN code with strangers that could eventually take advantage of it.

The same problem applies to personal assistant systems where questions and answers rely on external web-services that are out of the control of the user.

For this reason and under the perspective of privacy, we can argue that those approaches that do not rely on external services such as expert systems, some QA systems, model-based, and desktop facilities, are more secure for supporting users with sensible information.

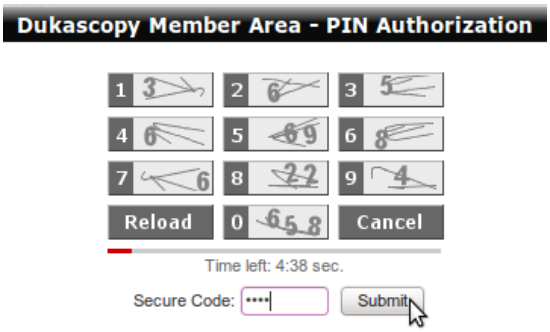


Figure 2.8: Authentication dialogue of a broker bank.

Explanations or answers dealing with all these previous aspects in the best possible way require to increase the cost of developing such a solution. Next section describes the problem of cost.

2.3.1.3 Cost

The lack of good help and support in most of the today's software is mainly due to a problem of **cost**. Software industry has become very competitive and one way to reduce the costs

of a product is to simplify documentation at the minimum level. In the previous work, the different reviewed solutions have also different cost levels. For instance, if we consider the classical approaches, the development of help systems relying on *expert systems* that have been detailed in section 2.2.1 requires the implementation of the the *inference engine* as well as the definition of the *knowledge base* for the specific application. In the same manner, *QA systems* require to implement the natural language interpreter as well as the information retrieval system. Model-based solutions that demand to adopt a specific frameworks like the Crystal framework in order to exploit the help system, also implies extra cost because of the time of integration and adoption of such frameworks by the developers, plus the time of implementing the help itself with the specific programming routines of each framework. On the contrary, model-based approaches using models that are already defined for the application, such as the modelisation of actions in Cartoonist, do not require almost any extra cost as the modelisation effort has already been done for the application anyway. In these cases, model-based solutions approaches present a low cost solution in comparison with previous alternatives.

Most of the last approaches presented in the state of the art show an improvement in the reduction of the associated cost. For instance, *Social-Network Based Systems* such as the one described in section 2.2.4 based on *Twitter*, reduce the cost of interpreting questions and composing answers or explanations because this work is done by users that are external to the application. With respect to the *Desktop facilities* as the one proposed by *Apple* that we have described, they are frequently embedded into the API of the underlying framework so the functionality comes without almost any cost for the developers as it is provided by the systems.

The next section analyses all the different criteria extracting a global conclusion about the different approaches.

2.3.2 Conclusion

Table 2.6 summarizes the previous discussion. Each approach shows different advantages and disadvantages. Considering that we want to cover a diversity of questions and explanation types, we must discard those approaches that are not suitable for a good *Coverage*. Due to the performance according to the *Quality* criterion, we must also discard Social-Network based systems as well as personal assistants. The three remaining approaches present equivalent results according to the presented criteria.

For those model-based approaches where the knowledge-base is contained into the same models that are necessary to build the application, we can notice a significant reduction of cost. Also, these systems provide a high *Quality* performance because they are usually reliable, i.e., their knowledge base is provided by experts or contained in models designed by experts of the domain.

	Coverage of the Questions	Quality of the Answers	Cost
Expert Systems	✓	3/4	
QA Systems	✓	3/4	
Model-Based	✓	4/4	✓
Social-Network Based	✓	0/4	✓
Personal Assistants	✓	0/4	✓
Recommender Systems		4/4	
Desktop facilities		4/4	✓
Avatars		2/4	

Table 2.6: Summary of supported criteria by approach.

To summarize the arguments:

Coverage As classical approaches are usually able to cover a larger set of questions, it could be worth to research what is the real coverage of questions of model-based approaches for help systems.

Cost In model-based approaches, the design models can nowadays be kept at runtime. Exploiting these models for explanations becomes then possible. If these design models

can be directly used to support users at runtime, this solution could dramatically reduce the cost of the resulting help systems, because the design models are already created during the development process.

Quality With regard to the quality aspects that we have considered, we can say that:

Availability As the models are kept at runtime, model-based answers are usually always available.

Reliability If the design models can be successfully used to support users at runtime, these models become the knowledge base of the help system. Thus, answers could be reliable because all the necessary information about the application is already captured into these models.

Security As the models are embedded into the application, the help system could deal with security related issues containing sensible information.

Time of response will be lower than other solutions relying on external services because the knowledge-base, i.e., the models used for explanation purposes, are embedded directly into the application.

Moreover, help facilities can be taken into account directly into the design process. Because they rely on the same models of the UI, the support facility is not something that is written later and added as an external entity of the system. The help facility is an integrated part of the system from the very beginning of the design of the UI.

Following all these reasons, this research explores how model-based help systems can achieve a good compromise between all the previous related work. In the next section we describe the problem space of this research, designed for covering the model-based facilities that have been described here. This problem space will help to better understand how the different model-based related works are positioned, and so to identify potential areas of improvement.

2.4 Focus on Model-Based Approaches

Our research explores how model-based explanations can contribute to better support end users when they interact with a user interface. In order to understand what are the contributions of this research with regard to the existent model-based approaches, this chapter sets the problem space. This problem space is based on three main areas that have been chosen according to the previous state of the art. After presenting the problem space in a first section, the second part of the chapter shows how different related works on model-based explanations are projected into the problem space, identifying key areas that have been studied in this research.

2.4.1 The QAP Problem Space

The QAP problem space (named QAP for its three main areas describing *Questions*, *Answers*, and *Properties*) is presented in figure 2.9. We have identified three main areas represented with external circles, each of them subdivided into different axes. The three main areas of the QAP problem space are described in the following:

1. The **questions** section represents the *input* of the help system, i.e., the way in which the user asks the system for information. It is named *questions* because, as we have previously seen in the chapter state of the art, questions are the most common way users normally use to request information. Questions can be asked in many different ways, for instance, using some sort of natural language or tooltips[8] among others. A question categorizes the form (presentation) and content (abstraction) of the user's request that the help system must address.
2. The **answers** section represents the output of the help system. As with questions, answers can be provided also in different forms, such as text, images or animations. Similarly to the questions section of the QAP problem space, an answer categorizes the form

[8] For a definition and examples of tooltips see `http://en.wikipedia.org/wiki/Tooltip`

Figure 2.9: The QAP problem space for support systems classification.

(presentation) and content (abstraction) of the support provided by the system to the user.

3. The **properties** section collects some features of help systems that are relevant in the context of this research. These properties are Extensibility, Dynamicity, and Initiative.

The subdivision of each of the two first areas, *questions* and *answers*, into a *Presentation* and *Abstraction* sections, is motivated by the three classification methods of explanation types identified by Gregor and Benbasat in [54]. These three methods are: *content*, *presentation format*, and *provision mechanism*. Contrary to Gregor and Benbasat, we propose a duality between questions and answers, being both of them subdivided into *Abstraction* and *Presentation*. In our problem space, the *Abstraction* category can be understood as the Gregor and Benbasat's *content* type, and our *Presentation* category belongs to the *presentation format* explanation type from the same authors. We will consider the *provision mechanism* later in the *Properties* section of the problem space.

Next sections detail each of the three main areas of the QAP problem space in detail.

2.4.1.1 Questions

The *presentation* and *abstraction* subareas in which questions are subdivided are detailed as follows:

Abstraction represents the nature of the request that the system needs to deal with. It can address questions about the *usage* of the system (as for instance, *How can I do this?*, or *Why this option is not enabled?*) and the *Design rationale* of the system (for instance, *Why these elements are ordered in this way? Why this message is big and red?*.

Presentation means how questions or user's requests are integrated into the user interface. The presentation can be either *intrinsic* or *extrinsic*. In the *intrinsic* systems the question is weaved into the user interface and it uses some of its elements for the creation of the query or questions. Left part of figure 2.10 shows an example of a help system presented in an intrinsic way. In the *extrinsic* systems, the question is formulated via

an external, non-integrated, and independent interface, that does not necessarily use elements of the user interface to specify the request. For instance, manuals and online help systems such as the one shown in the right side of figure 2.10 are examples of *extrinsic* ways of asking for information.

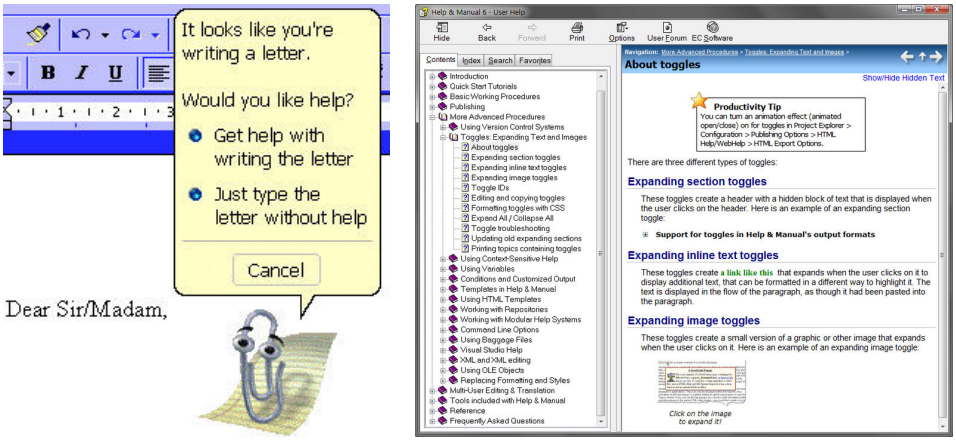


Figure 2.10: Examples of different types of help according to the Presentation axis. *Left*: Intrinsic help (Clippy from Microsoft Word©). *Right*: Extrinsic help (An online help dialogue presented outside the application that contains all the supportive information ordered by categories).

The motivation for the subdivision into *intrinsic* and *extrinsic* axes comes from for different studies as for instance in [135], where Sheneiderman et al. also defines the degree of integration in the interface (from less to more integrated) as

- Online documentation and tutorial: independent interface, even possibly developed by a different company
- Online help: integrated into the interface, separate window usually invoked from a "help" button
- Context-sensitive help: a kind of online help that is obtained from a specific point in the state of the software, providing help for the situation that is associated with that state.

With regard to our taxonomy, the first degree identified by Shneiderman et al. referring to independent interfaces is equivalent to our *extrinsic* axis, while the second degree corre-

sponding to the integration inside the interface of the application is similar to our *intrinsic* axis. In relation to context-sensitive help, in our taxonomy we exclude elements related to how the answer or explanation is computed by the system because rather than focus on how the explanation is computed, (i.e., if for instance the state of the application is considered when computing the answer or not), we prefer to focus on what type of information the explanation is about. The reason is that as this problem space is intended for classifying model-based related solutions, we estimate that model-based approaches can most of the time know the state of the application in one way or another. For instance, a *tooltip* is considered to be an example of context-sensitive help. According to our taxonomy, *tooltips* are related to questions about *Usage*, and in particular, they answer questions about the *Representation* of the user interface (*What is this for?*).

The *Abstraction* axis is divided into *Usage* and *Design rationale*. First, help systems for end-users have been traditionally oriented to questions related to the use of the application. Second, designers have also been supported into the design process through different design notations. For instance, conception decisions, specially those related to the user interface, are addressed at design time by the designers. Different design rationale notations have been traditionally used by the designers to keep track of these design decisions, and some of them as for instance QOC [84], model these decisions in the form of questions, it seems clear that this kind of supportive information can be also useful for end-users under certain situations.

Next section covers the description of the answers section of the external circle.

2.4.1.2 Answers

Similarly to questions, answers are also subdivided into *presentation* and *abstraction* subareas, both of them described in the following:

Presentation means how answers -i.e., information support- are integrated into the user interface. As for questions, the presentation of the answer back to the support can be either *intrinsic* or *extrinsic*. An *intrinsic* answer is weaved into the user interface and it can even use some of its elements for presenting the information support to the user. In

extrinsic answers, the answer presented via an external, non-integrated, and independent interface, that does not necessarily use elements of the user interface to provide the support.

Abstraction represents the type of knowledge desired in the answer. The possible types are *Representation*, *Structure*, *Task-Concepts*, and *Functionality*.

Representation indicates that the type of answer is related to the physical representation of the user interface. These kind of answers are normally addressed to widgets or elementary interactors of the user interface, for instance *What is this for?*.

Structure represents answers about the way in which the parts of the system or the user interface are arranged or organized. Questions related to navigation issues are classical examples of this axis (*Where is...?*).

Task-Concepts indicates answers about goals and their related concepts. This axis covers traditional questions about goals such as *How do I do this?* or *What can I do now?*

Functionality describes answers related to the functional core of the application. For instance, *What happened?*

The subdivision into Representation, Structure, Task-Concepts and Functionality is inspired from several previous works.

The first one is the Foley and Van Dam's levels of abstraction of interactive systems [41]. Here, the authors identified four different levels: the Conceptual level -how the user views the system in terms of concepts and tasks-, the Semantic level -describes "the meanings conveyed by the user's command input and by the computer's output display", the Syntactic level -it defines "how the user's actions (units, words) that convey semantics are assembled", and the Lexical level -dealing with device dependencies and specific syntax of the user's request". In comparison with the Foley and Van Dam's levels of abstraction, we keep the conceptual level specifying explicitly both tasks and concepts in the *Task-Concepts* category, and we group the semantic, syntactic and lexical levels into the *Representation* category.

The second work inspiring the decomposition of the *Abstraction* category into four different axes is the "Taxonomy of user documentation, online help, and tutorials" proposed by Shneiderman et al. ([135]. Shneiderman and colleagues state that the domain covered by a help system can refer to:

- Description of interface objects and actions (syntactic)
- Sequences of actions to accomplish tasks (semantic)
- Task-domain-specific knowledge (pragmatic)

Here, we consider the syntactic level as our *Representation* category. The semantic and pragmatic levels are grouped into the *Concepts-Tasks* previously described.

This subdivision into these four different axes is also motivated by some architectural patterns. The first one is the Presentation-Abstraction-Control (PAC) architectural pattern [26]. PAC is an interaction-oriented software architecture that divides a system into three different components. Each of these components is responsible for a specific aspects of the system. The abstraction component retrieves and processes the data, the presentation component formats the visual and audio presentation of data, and the control component handles things such as the flow of control and communication between the other two components.

The architecture of PAC is quite similar to that presented in the Model-View-Controller (MVC) architectural pattern[9].

Both architectural patterns make distinction between the user interface (the *Presentation* in PAC or *View* in MVC) and the functional core of the application (the *Control* in PAC or the *Controller* in MVC.

Other architectures, specially those coming from model-based approaches of user interfaces, have also motivated the subdivision of the *Abstraction* section into four different axes. In particular, most of model-based approaches of user interfaces separate the UI itself from the functional core, and subdivide the UI into different levels of abstraction. As an illustration, we consider here the Cameleon Reference Framework [17]. This framework defines a UI

[9] Interested readers can refer to [123] for a deeper discussion (chapter nine, *"Interaction-oriented Software Architectures"*).

in terms of four different levels of abstraction[10]: Task and Concepts level, Abstraction level, Concrete level, and Final UI level. In this framework, the Abstraction level groups tasks in a platform independent way, structuring them in a platform dependant way at the Concrete level. Thus, this level represents for instance the widgets and their arrangement in a Graphical User Interface. The last level called Final UI represents the code that implements the widgets in a specific language.

In our problem space, the information about the Representation and the Structure of the user interface is shared by the Abstraction and Concrete levels of the Cameleon framework. We keep the Tasks and Concepts level as well as the Functional core, which is present in most of the model-based approaches that we have reviewed. Table 2.7 summarizes the discussion about the different works in which these axes are based.

2.4.1.3 Properties

The properties section contains the following axes:

Dynamicity indicates if the information provided by the help system to the user is generated at runtime, i.e., computed directly by using some source of knowledge. Answers or explanations that are not dynamically generated rely on predefined support that cannot be modified once the application is running, or in other words, that needs to be rewrit-

[10]Because of the relevance of the Cameleon Reference Framework for this research, it is described in detail in chapter 3

	Foley	Schneiderman	Cameleon	PAC	MVC
Representation	Semantic, Syntactic Lexical	Syntactic	Abstract Concrete	Presentation	View
Structure	Semantic, Syntactic Lexical	Pragmatic	Abstract Concrete	Presentation Control	View Controller
Tasks-Concepts	Conceptual	Pragmatic	Task-Concepts	Abstraction Control	Model Controller
Functionality				Control	Controller

Table 2.7: Works contributing to the *Answer* axis.

ten and updated by hand at design time.

Initiative represents how the action of providing support is started. This axis represents those help systems that are able to initiate the support by themselves, rather than waiting for requests from the users.

Extensibility means whether the support provided by the help system can be improved in some manner, for instance by adding annotations, or new sources of knowledge to the system.

The motivation for these properties rely on the following literature. First, the *Initiative* axis have been also identified for instance by Gregor and Benbasat in the aforementioned work. The authors describe three types of mechanisms to provide explanations regarding the time of intervention: user-invoked, automatic, and intelligent.

Schneiderman et al. distinguish however between:

- Before starting (quick guide, manual, and tutorial)
- At the beginning of the interaction (getting started, animated demonstration)
- During the task (context-sensitive, either user or system initiated help)
- After failure (help button, FAQs)
- When the user returns the next time (start-up tips)

"Before starting" help such as guides or manuals is out of the scope of this research. Some of the rest of the properties are initiated automatically by the system (getting started or start-up tips) whilst the rest demand the intervention of the user. The Initiative axis helps to identify and distinguish between both types of help system.

We consider both the *Extensibility* and *Dynamicity* axes as important properties because of the criteria previously described in chapter 2. In particular, the problem of **cost** requires help systems that are generated and not written at design time. The **cost** factor also demands that the help system should be easily extended without rewriting the whole information support, for instance, for adding new information or new types of explanations.

Next section explains how to read the different values of each axis.

2.4.2 Reading the QAP Problem Space: Values of the Axes

All the axes of the problem space illustrated in figure 2.9 have binary values, i.e., a true value whether the characteristic of the axis is supported or not by the help system under study. A true value (i.e., a help system supporting the characteristic of the axis) is represented at intersection between the supported axis and the exterior circle. A false value is represented at the intersection between the axis and the interior circle. According to this, a help system is represented into the QAP problem space by finding the different values of the help system according to each of the characteristics of the axes. For instance, figure 2.11 shows an excerpt of a help system under study that has been mapped into the problem space. This hypothetical help system is able to provide answers about the *Representation* and *Structure* of the system, but it does not have support for answers related to *Tasks-concepts* and *Functionality* of the system.

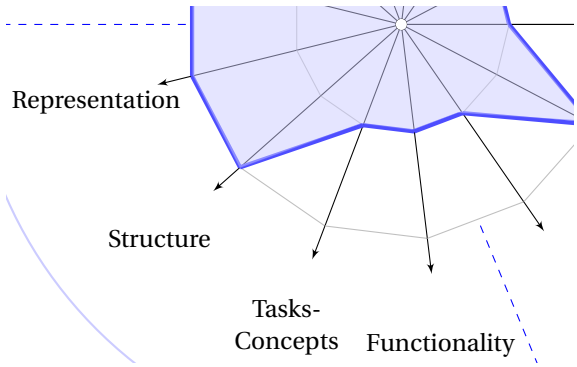


Figure 2.11: Hypothetical system under study supporting answers about the Representation and Structure of the system, and without support for tasks-concepts and functionality related answers.

Next section illustrates some related work from the state of the art. The main objective of the QAP problem space is to compare the result of this research with other model-based solutions. For this reason, we map into the problem space only those works that have been previously identified as model-based approaches or model-based related for providing explanations, so we can better identify gaps and areas of improvement for this research, using the

promising model-based approach of user interfaces.

2.4.3 QAP Problem Space and Related Work

This section maps some works from the section State of the art into the problem space.

2.4.3.1 Crystal System

The Crystal application framework provides an architecture and interaction techniques that allow programmers to create applications that let the user ask a wide variety of questions about why things did and did not happen in the user interface, and how to use the related features of the application without using natural language [100]. The "Why" and "Why not" questions supported by Crystal are related to user's actions. Crystal supports then questions about the *Usage* or the system (axis Question-Abstraction-Usage).

Crystal uses a Command Object model [10] to implement all of the actions. The commands the user executes are stored on a command list which serves as a history of all the actions that have been taken, and used later for answering "Why" and "Why not" questions about user's actions. As these commands represents the interaction of the user with the system, they model the tasks the user is performing. For these reason, these specific ques-

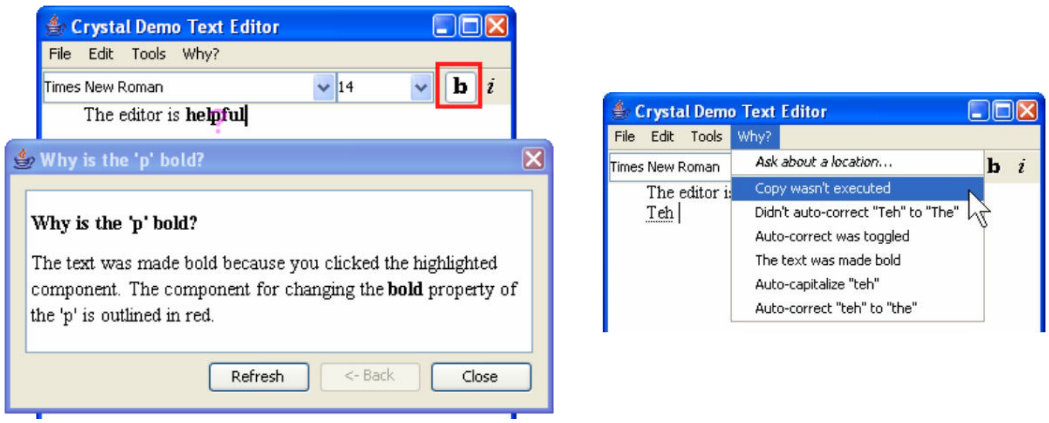


Figure 2.12: Left: Crystal answers "Why is the 'p' bold?" by highlighting the relevant user interface controls. Right: The "Why?" menu. Images adapted from [100].

Figure 2.13: Myers' Crystal help system.

tions about “Why” and “Why not”provide answers related to the *Tasks-Concepts* axis (Answer-Abstraction-Tasks-Concepts). Crystal also supports asking “How can I” questions (see [100] for more information). The type of answers is also related to the *Tasks-Concepts* axis.

The way in which the support is requested is by an integrated help menu or by directly pressing the F1 key. Thus, the presentation of the question is *intrinsic* as it is embedded into the same user interface of the application.

The answer is *extrinsic* because the message support is shown in a separated window, but it supports *intrinsic* information as well because answers can highlight the relevant elements of the user interface as shown in figure 2.12.

The Crystal framework is not easily extensible and only a predefined set of question types are supported (false value of the *Extensibility* axis). However the answers are generated dynamically from the models (true value of the *Dynamicity* axis).

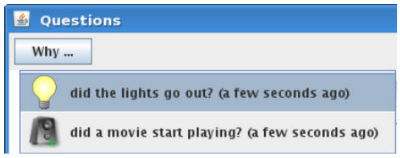
Finally, Crystal needs the user to request for support, so the *Initiative* axis has a false value.

The Crystal system is mapped into the QAP problem space in figure 2.13.

2.4.3.2 PervasiveCrystal System

The PervasiveCrystal [153] system keeps the idea of the previous Crystal framework but adapts it to pervasive environments. The authors state that there is no pervasive computing frameworks available that supports why and why not questions about the behaviour of the system. Moreover, existing desktop implementations such as the previous Crystal system cannot be easily integrated into pervasive computing frameworks, since the assumptions underlying these implementations rarely hold in pervasive computing. For instance, pervasive environments usually rely on multiple machines from which events originate. As an example, consider the situation where the user starts playing a movie on the TV and the lights go out. The system involves at least, the TV, the lights of the room, the sensors, and the system processing such events.x

PervasiveCrystal captures the behaviour of an environment in a model built up from rules that connect actions and events. These rules are implemented according to the Event - Con-



(a) The why menu.

(b) An answer to a why question.

Figure 2.14: The PervasiveCrystal system ([153]) answering a Why question.

dition - Action (ECA) paradigm and extended with inverse actions to be able to answer both "Why" and "Why not" questions about the behaviour of the system.

As PervasiveCrystal is able to answer "Why" and "Why not" questions, following the same reasoning described for the Crystal system, we can conclude that PervasiveCrystal covers the *Usage* axis (Question-Abstraction-Usage). PervasiveCrystal is able to answer questions about the behaviour of the system (see figure 2.14) so it supports the *Functionality* axis into the Answer-Abstraction section of the problem space. As the PervasiveCrystal system also models the actions of the user or tasks, it also supports the *Tasks-Concepts* axis positively.

The request for help is performed by users through an integrated help menu that able them to ask for explanations. This means that questions are weaved into the user interface in an *intrinsic* manner.

As in the previous Crystal system, explanations can be provided in both *intrinsic* and *extrinsic* ways.

The PervasiveCrystal system share the same properties of its predecessor. It is not easily extensible for supporting more questions of a different type, the initiative of providing support needs an explicit request by the user (the system does not propose any help), but on the other hand, all the answers are dynamically computed and generated at runtime.

Figure 2.15: Vermeulen's PervasiveCrystal system.

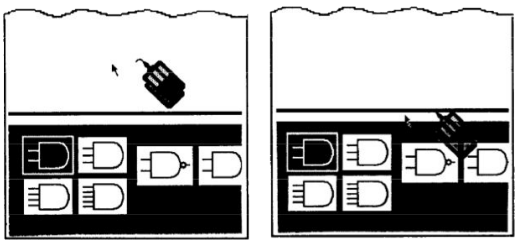


Figure 2.16: The Cartoonist system ([141]) in action: "the animated character of the mouse is shown trailing the cursor which is being moved to click on the *Create a NAND gate* command icon."

The PervasiveCrystal system is mapped into the QAP problem space in figure 2.15.

2.4.3.3 Cartoonist System

The Cartoonist system [141] automatically generates help for explaining how to accomplish tasks. The explanations given by Cartoonist are provided in the form of animations. Cartoonist employs a kind of task model (in the form of user's actions) for generating such animations. The animations constructed by Cartoonist show how to invoke the commands of an application. As shown in figure 2.16, the mouse pointer is explicitly represented by a graphic. This graphic moves around the user interface, picking the objects from a panel of elements, and setting them up to complete specific tasks.

The questions supported by Cartoonist are then related to the *Usage* of the user interface. As the answers supported by Cartoonist cover information about "How can I ..." questions, this systems relies on the *Tasks-Concepts* axis.

The way in which questions are asked to the system is by a simple external dialogue in which the user can type the name of the task that will be shown by the system. This means that the presentation of the questions is then *extrinsic*. The answer is an animation of a "three-button mouse" icon that performs the task directly into the user interface. As the answer uses the controls of the user interface, its presentation belongs to the *intrinsic* axis.

The Cartoonist system is not extensible because it only supports the question type "How can I" for what the system has been designed.

Figure 2.17: Sukaviriya's Cartoonist help system.

The action is initiated by the user so the system does not support the axis *initiative*.

As we have shown, Cartoonist creates animations dynamically at runtime so it supports the *Dynamicity* property represented in the *Dynamicity* axis.

Figure 2.16 shows how the Cartoonist system is mapped into the QAP problem space.

2.4.3.4 Intelligibility Toolkit

Figure 2.18: Image and description from [79]. This explains "Why" the application inferred "Breakfast". The evidences are indicated by the area of bubbles around the corresponding sensors in the floorplan.

Lim et al. propose [79] the Intelligibility Toolkit for asking several questions about user interfaces in the context of ubiquitous computing. According to [79], "The Intelligibility Toolkit makes it easier for developers to provide many explanation types in their context-aware applications. This ease also allows developers to perform rapid prototyping of different explanation types to discern the best explanations to use and the best ways to use them."

The Intelligibility Toolkit tries to make context-aware applications intelligible[11] by "automatically providing explanations of application behavior" [79]. To this end, the toolkit pro-

[11] For a definition of the term "intelligibility" see [8]

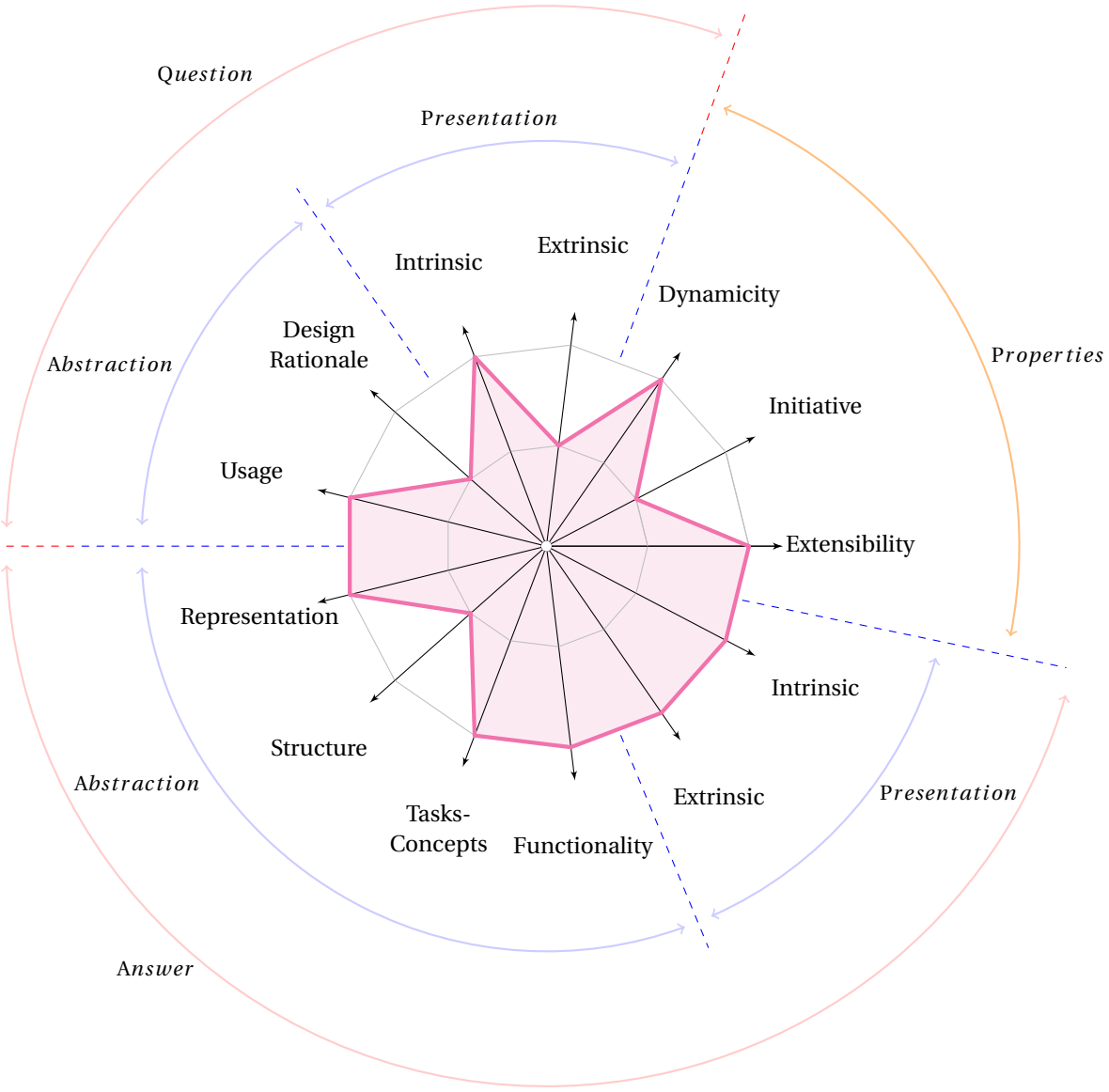


Figure 2.19: Lim's Intelligibility Toolkit.

vides automatic generation of eight explanation types (Inputs, Outputs, What, What If, Why, Why Not, How To, Certainty) for four different decision model types (rules, decision trees, naïve Bayes, hidden Markov models). All the Wh- explanation types along with the Outputs and Certainty types are related to the behaviour of the application. For instance, "What if" explanations "allow users to speculate what the application would do given a set of user-set input values", and the "Why" explanations inform users "why the application derived its output value from the current (or previous) input values"[79].

An example of this is shown in figure 2.18. The figure shows a Home Activity Recognizer using the Intelligibility toolkit. It shows a map or floorplan indicating why the recognised activity is Breakfast. The explanation is a floorplan visualization of the evidences in the last 5 minutes:

Sleeping → Toilet → Toilet → Breakfast → Breakfast

For instance, the Hall Bedroom Door being open is a strong indicator of inferring the sequence. The microwave is another strong indicator (biggest bubble in top right corner).

The ability of the toolkit for supporting behaviour questions means that it supports the axis *Functionality*. And because the toolkit is able to provide information about the input values of the system through the Inputs explanation type, it supports the *Representation* axis as well. The "How to" explanation type makes the system supports the *Task-Concepts* axis.

All these different explanation types provided by the toolkit are related to the usage of the system, so it supports the *Usage* axis.

The toolkit is extensible to support new explanation types and model types all related to the behaviour of the application. For this reason, the axis *Extensibility* is supported.

This system supports requests by the user in a *intrinsic* way, and the presentation of the answers can be represented in both *intrinsic* and *extrinsic* configurations, because the Intelligibility Toolkit provides different presentation formats so developers can chose the more suitable presentation format for the explanation in their applications.

Finally, the toolkit requires that the user demands for support (*Initiative* property not supported), and the explanations are generated dynamically with explanation generation algo-

rithms based on the four aforementioned decision models (rules, decision trees, naïve Bayes, hidden Markov models).

Figure 2.19 shows how the proposed toolkit is mapped into the QAP problem space.

2.4.4 Overlapping Analysis

Figure 2.20 shows the resulting overlapping of the precedent works. This overlapping gives a global overview of where most of the works have currently focused for supporting users. Three main areas of interest have been identified as they are uncovered by the reviewed literature.

The first one concerns questions regarding the *Design rationale* of the user interface. The axis is not covered by any work. We did not identify any previous research able to provide questions about the design rationale of the user interface, that can help users to better understand "Why" the user interface is the way it is. Design rationale questions can also be potentially useful for specific learning purposes in, for instance, a user interface design training course. Design rationale information can be useful as well not only because some interactions can be directly affected by design decisions on the UI, but also because potential end-user programmers can probably take benefit of accessing this information at runtime.

To address this particular area, we will try in this research to enrich help systems, i.e., the types of questions that users can ask, with information related to the design decisions that are made at design time.

The second uncovered axis is related to the *Initiative*. Initiative has widely covered by agents as shown in the previous chapter. For this reason, we will not directly address this issue from a model-based point of view but, instead, we will study what types of questions can take benefit from active help systems, to open other potential research questions.

The third area of interest is the *Structure*. Information about the structure of the user interface is not usually provided. Structural information about the user interface can help to explain the way in which the parts of the system or the user interface are arranged or organized, so for instance, navigational questions could be finally supported. In the context of this research we will explore how we can extract structural information to better support the user

Figure 2.20: Overlapping the related work. Identifying potential areas of interest.

with this type of information.

We also appreciate in figure 2.20 that some systems allow developers to choose the *Presentation* of *Answers* either in a *Intrinsic* or *Extrinsic* way. However, this is not the case for *Presentation* of the *Questions*, imposing one of both alternatives to the design of the user interface. We will explore how to overcome this limitation so designers and developers can fully customize how both questions and answers are integrated into the system. This will bring the designers the possibility of choosing their preferred presentation mode, or even combining both of them at the same time.

Finally, the figure 2.20 shows that there is an implicit problem of **unification**. In fact, each of the reviewed model-based works such as Crystal or PervasiveCrystal address a specific type of questions, but we are not aware of any work that currently unifies different types of explanations at the same time. An approach for asking questions either for the usage or design rationale in a homogeneous way becomes necessary for supporting different question types simultaneously. In the same way, providing different types of answers require to uniform the way in which these answers are computed.

The next section summarizes the chapter.

2.5 Synthesis

First, the chapter starts by reviewing what is an explanation, how explanations have been considered in the Theory of Explanation of the Philosophy of Science, the role of explanations in the Erotetic Logic, and how the concept of Structural Explanations has emerged.

Second, explanations have been covered by a large amount of fields in computer science. The chapter reviews the different approaches that have contributed to supporting users through explanations, describing and providing examples for expert systems, question answering systems, model-based systems, Social-Network Based Systems, Personal assistants, Recommender Systems, Desktop facilities, and Avatars.

Third, the approaches are compared through a set of different criteria, that help the au-

thors to consolidate model-based as a promising research approach for providing explanations to users.

Four, a more deep study is conducted through the QAP problem space specially oriented to model-based systems. The QAP problem space helps to identify the potential areas of improvement by mapping the previous related work into the problem space, and analysing the overlapping of their areas.

With the identification of the approach and the research axes that will guide this research, next chapter will present all the elements that are necessary to put into practice the model-based approach of user interfaces, its models, and other relevant concepts.

3

Foundations

> *If you have built castles in the air, your work need not be lost; that is where they should be. Now put the foundations under them.*
>
> Henry David Thoreau, Walden,

In our research we explore whether the model-based approach of user interfaces is suitable for helping users in understanding the user interface. In particular, we investigate if it is possible to generate explanations based on the models of the user interfaces made at runtime, and whether these explanations are suitable for end users. Thus, the models of the user interface become the knowledge-base of the help system. To this end, we need to understand how the model-based approach is used in the development of UIs and what these models are. This is the purpose of this chapter.

The chapter firstly reviews the relevant concepts and terms used by the model-based approach. Then, we describe the model-driven engineering of user interfaces in a second section. Finally, as quality is an important aspect to provide design rationale explanations, we also review different quality models at the end of the chapter.

3.1 Model-Driven Initiatives

This section introduces the model-driven concepts necessary to understand this research. The section starts with a definition of the different model-driven initiatives (Model-Driven Architecture (MDA), Model-Driven Development (MDD), Model-Driven Engineering (MDE) and Model-Based Engineering (MBE)) through a brief review of their history. Then, the section provides all the necessary definitions.

3.1.1 Model-Driven Initiatives: A Brief History

In conceptual modelling, models represent concepts or entities and the relationships between them.

It was in 1976 when Peter Pin Shan Chen proposed [23] the Entity-Relationship Model (ER), which characterizes the concepts of Entity and Relationships. In words of the author, "The ER concept is the basic fundamental principle for conceptual modeling." ([1]). ER was progressively extended. For instance the version of the Enhanced Entity–Relationship model by Elmasri and Navathe [36] includes all the concepts introduced by the ER model, adding the notions of subclass and superclass, with the related concepts of specialisation and generalisation [36]. These concepts were the foundation of the Unified Modeling Language (UML), firstly published in 1997, after the unification of the three methods of their authors, namely the *Object-Oriented Analysis & Design* (OOAD) by Grady Booch [13], the *Object Modelling Technique* (OMT) by James Rumbaugh [127], and the *Object-oriented Software Engineering method* (OOSE) by Ivar Jacobson [64].

ER and UML are considered the main roots of Model-Driven Architecture, but other computer science disciplines have also contributed to MDA. For instance, the formal concept of Object in programming was introduced in the 1960s in the language Simula 67. However, the first Object-Oriented language, *Smalltalk*, was developed at Xerox PARC in the 1970s. Smalltalk is considered to be the first Object-Oriented language because it was designed to be a fully dynamic system in which classes could be created and modified dynamically rather

than statically as in Simula 67.

Nowadays there is a strong relationship between object-oriented languages and UML. UML is today used as the standard model-based language for software development. It has continued to evolve (UML 2.0 dates from 2005) becoming the standard model-based language for software development. The MDA initiative is strongly related to UML, and it is the first model-driven initiative proposed by the Object Management Group (OMG) in 2001 (see [109, 93]).

Figure 3.1[1] shows different initiatives that are based on models and the relationships between them. The central and older initiative is the Model-Driven Architecture which is founded on the notion of conceptual modelling.

Next section details the different layers shown in figure 3.1, from the most central, Model-Driven Architecture or MDA, to the most general, Model-Based Engineering or MBE.

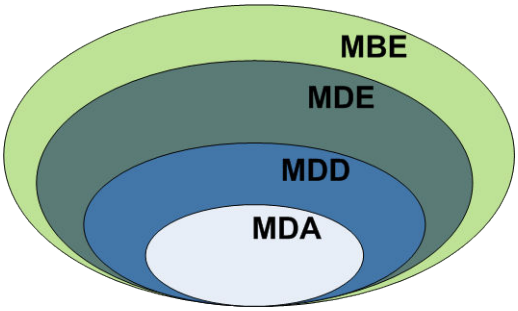


Figure 3.1: Relationship between model approaches followed in this research.

3.1.2 Model-Driven Architecture

The current official MDA definition according to [110] is the following:

[1] Image taken from
http://modeling-languages.com/model-based-engineering-vs-model-driven-engineering-2/

> MDA is an OMG initiative that proposes to define a set of non-proprietary standards that will specify interoperable technologies with which to realize model-driven development with automated transformations. Not all of these technologies will directly concern the transformations involved in MDA. MDA does not necessarily rely on the UML, but, as a specialized kind of MDD (Model Driven Development), MDA necessarily involves the use of model(s) in development, which entails that at least one modelling language must be used. Any modelling language used in MDA must be described in terms of the MOF language, to enable the metadata to be understood in a standard manner, which is a precondition for any ability to perform automated transformations.

The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns. According to this and as stated in [109], MDA provides an approach for, and enables tools to be provided for:

- specifying a system independently of the platform that supports it
- specifying platforms
- choosing a particular platform for the system
- transforming the system specification into one for a particular platform

MDA, as well as all the OMG initiatives, follows the principle that "everything is a model" as stated in [11]. The next section describes the notion of model and its related concepts.

3.1.2.1 Models

Several definitions of model [109, 94, 68] are summarized in [38]. For instance, in the context of the UML standard, the term model is defined as following:

> A model is an abstraction of a physical system, with a certain purpose.

Other authors define the term in function of a language [68]:

> A description of (part of) a system written in a well-defined language.

The OMG also clarifies in [109] that "a model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language."

Figure 3.2 shows an example of a model, where two different entities, *Persons* and *Cars*, are related between them through the *Owns* relationship.

A model can be considered as a sentence (or just a word) of the modelling language in which the model is expressed [137]. When we talk of this modelling language, we are in part referring to the meta-model of the model. This concept of meta-model is reviewed in the next section.

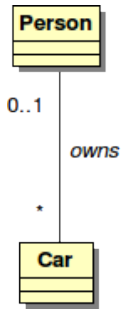


Figure 3.2: Example of model. The entity *Person* can own an unlimited number of *Cars*.

3.1.2.2 Meta-Models

The following definition is given in [37]:

> A meta-model is a model of a modelling language.

Thus, meta-models are a meaning for reasoning in terms of the modelling language. Favre stated in [37] that:

> A meta-model is a model of a set of models.

Figure 3.3 shows an example of a meta-model. The model previously presented in figure 3.2 conforms to this meta-model.

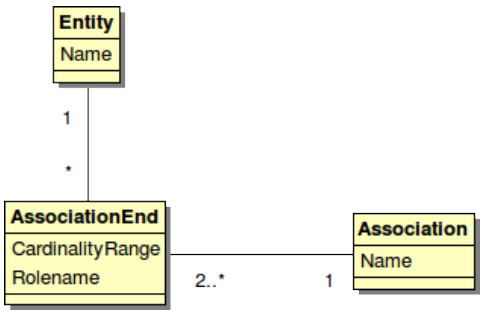


Figure 3.3: Example of meta-model. An *Association* is related to *Entities* thorugh *AssociationEnds*.

In the same manner that models are defined in terms of meta-models, meta-models are defined in terms of meta-meta-models, that are introduced in the next section.

3.1.2.3 Meta-Meta-Models

We can reuse the first definition of meta-model to define the concept of meta-meta-model as "the model of the modelling language of the meta-model", or simply:

> A meta-meta-model is a model of a set of meta-models.

To avoid an infinite number of "meta levels", meta-meta-models are said to be self-describing or *reflective*, meaning that they can be recursively defined by themselves.

Figure 3.4[2] describes graphically the different levels of abstraction and their relationships as defined in the Model-Driven Architecture.

[2] Image from the ATL reference manual.

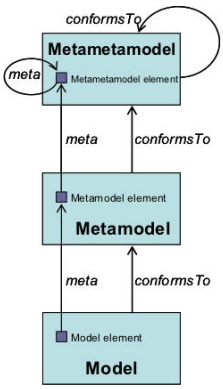


Figure 3.4: Model-Driven Architecture.

3.1.3 Four-layers architecture

OMG designates four different levels named with an M and the number of the level. These levels are summarized in figure 3.5. These four levels determine what is called the "Four-layers meta-modelling pyramid". Each of the levels includes one of the previous model, meta-model, and meta-meta-model concepts already presented. Plus, the pyramid shows the instances of the models or objects at the base.

The description of the levels is as follows:

M3 : Layer containing the meta-meta-models, described in terms of themselves because of the reflective property.

M2 : Layer containing the meta-models (for example, UML elements such as Classifiers, Attribute, and Operation, or definitions of a any modelling language).

M1 : Layer containing the models (for example, a UML class representing vehicles).

M0 : layer containing the objects of the application (for example, an instance of the class vehicles representing the car with license plate 12345).

All these four different layers become specially important when considering transformations. The next section describes the concept of model transformations in detail.

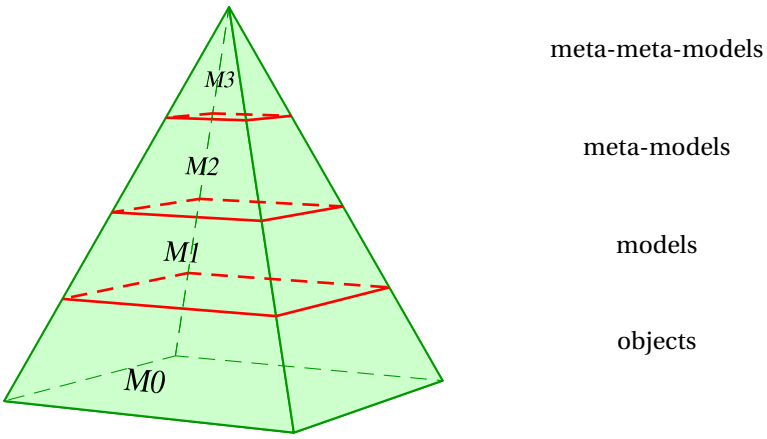


Figure 3.5: Four layers pyramid showing each of the OMG levels.

3.1.3.1 Model Transformations

Model transformations or simply transformations, are defined in [109] as:

> The process of converting one model to another model of the same system.

Transformations are explicitly represented in MDE, becoming first order elements of the MDE initiative as models or meta-models. They are normally defined as a set of rules that describe how one or more source models are transformed into one or more target models. These rules are defined in terms of the meta-model. For instance, when the transformation transforms models, its rules are defined in terms of meta-models. In the same way, when the transformation transforms meta-models, its rules are described in terms of meta-meta-models. Transformations are a key aspect in MDE because, as they are defined in terms of the modelling language (or meta-model), they can be reused for all the models conforming to the same meta-models. For instance we could build a transformation to transform an UML model into java code, and reuse this transformation for every UML model.

As transformations are first order elements, they are models as well. In fact, a transformation is defined in terms of a transformation language (meta-model). At this point it is possible to write transformations that transform one or more source transformations into one or more

target transformations. This special case of transformations are called High Order Transformations.

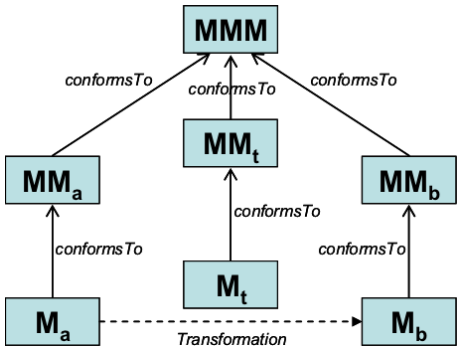


Figure 3.6: Model transformation.

Figure 3.6 summarizes the full model transformation process. A model *Ma*, conforming to a meta-model *MMa*, is here transformed into a model *Mb* that conforms to a meta-model *MMb*. The transformation is defined by the model transformation model *Mt* which itself conforms to a model transformation meta-model *MMt*. This last meta-model, along with the MMa and MMb meta-models, has to conform to a meta-meta-model *MMM*.

Figure 3.7 provides an example. The transformation *UML2Java* converts a UML diagram into java code. The *M-UML* box represents the UML diagram model to be transformed, that conforms to the UML meta-model (*MM-UML*) or UML language. The transformation generates a java model named *M-Java* that holds the information for the creation of Java classes, such as package references, attributes or methods. This model conforms itself to the Java meta-model represented by *MM-Java*. The designed transformation, which is expressed by means of a transformation language (the ATL language in this case), conforms to the *ATL* meta-model. In this example, the three meta-models, *MM-UML*, *MM-Java* and *ATL* are expressed using the semantics of the *Ecore* meta-meta-model.

Transformations in MDA play an important role. They transform one MDA model into another. MDA defines three types of models: Computation Independent Model (CIM), Platform Independent Model (PIM), and Platform Specific Model (PSM). These concepts are explained next.

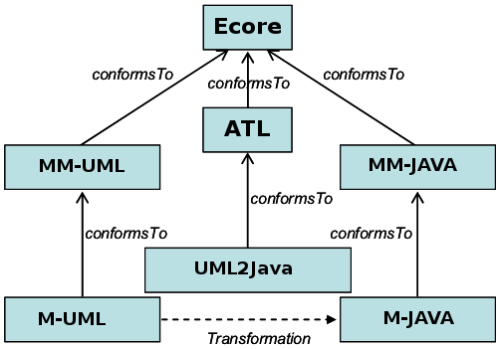


Figure 3.7: Example of a model transformation. Generation of java code from UML through an ATL transformation.

3.1.3.2 MDA Models

MDA specifies the following three default models of a system:

Computation Independent Model (CIM) A CIM is also often referred to as a *business* or *domain model*. It presents exactly what the system is expected to do. It is completely independent of the system.

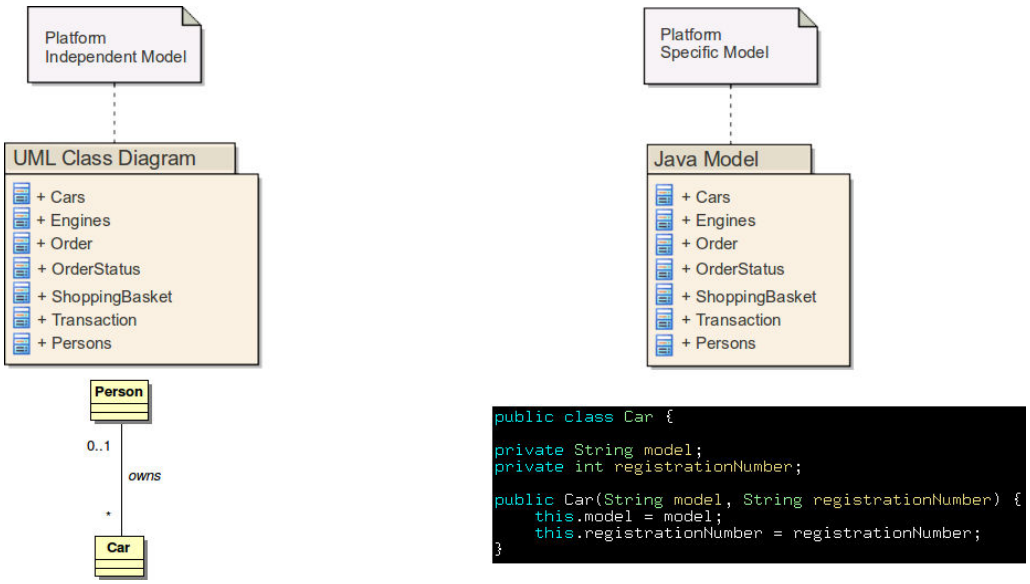


Figure 3.8: PIM model (left) and PSM model (right).

Platform Independent Model (PIM) A PIM exhibits a sufficient degree of independence so as to enable its mapping to one or more platforms. This is achieved by defining a set of services in a way that abstracts out the technical details.

Platform Specific Model (PSM) A PSM combines the specifications in the PIM with the details required to stipulate how a system uses a particular type of platform.

Figure 3.8 illustrates an example of the previous models. On the left side, a platform independent model is represented with the package containing the UML class diagram of the application. On the left side, a platform specific model is represented with a package of the java model of the same application. The relationship between both models is described in the next section.

3.1.3.3 The MDA Process

A complex system may consist of many interrelated models. The two key concepts of MDA are models and transformations. The general pattern between them is illustrated in figure 3.9:

Figure 3.9: General pattern of the MDA process.

As an example, consider the models previously illustrated in figure 3.8. The application model of the car shopping website is the source model, whereas the website on the right side is the target model. In figure 3.10, the PIM model is transformed into the resulting PSM model represented by the car shopping website.

3.1.4 Models, Meta-Models, and Meta-Meta-Models

The MDA definition relies on the term MDD or Model-Driven Development. According to this and as stated by Jean Bezivin in [11], MDA may be defined as “the realization of MDE principles around a set of OMG standards like MOF, XMI, OCL, UML, CWM, SPEM, etc.”. MDA is then the OMG’s particular vision of MDD and thus, it relies on the use of OMG standards.

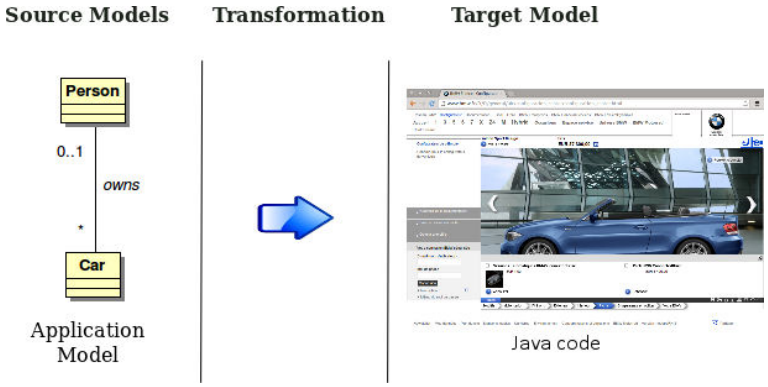


Figure 3.10: Example of the MDA process. Two PIM source models are transformed together to generate a single PSM model.

Therefore, MDA can be regarded as a subset of MDD.

Model-Driven Development is introduced in the next section.

3.1.5 Model-Driven Development

MDD has been formalized in 2003 ([95]) as follows:

> Model-Driven Development is simply the notion that we can construct a model of a system that we can then transform into the real thing.

MDD is then a development paradigm where the primary artefacts of the development process are the models. Usually in MDD, the implementation is (semi)automatically generated from the models.

Contrary to MDA, MDD does not adhere to any of the OMG standards as MDA does. MDA is usually considered as the OMG's particular vision of MDD.

According to [131], the main objective of MDD is to increase productivity, maximizing compatibility between systems, simplifying the design process and promoting communication between individuals and teams working in the system. MDD can be considered as a subset of MDE, which is defined in the next section.

3.1.6 Model-Driven Engineering

In 2006, three years after the definition of MDA, the concept of MDE was characterized in [128] by Douglas C. Schmidt as "an approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively."

The standard definition[3] of MDE is:

> Model-driven engineering (MDE) is a software development methodology which focuses on creating and exploiting domain models (that is, abstract representations of the knowledge and activities that govern a particular application domain), rather than on the computing (or algorithmic) concepts.

Contrary to MDD, MDE goes beyond of the pure development activities and encompasses other model-based tasks of a complete software engineering process such as the model-driven reverse engineering of legacy systems.

MDE is a considered to be a subset of MBE, which is defined in the next section.

3.1.7 Model-Based Engineering

The term Model-Based Engineering (MBE), also refered in the literature as Model-Based Development (MBD), is currently interpreted and approached in many different ways. There is however a general consensus in the literature that defines the term as a softer version of MDE, in which models play an important role although they are not necessarily the key artefacts of the development. In other words, models do not drive the development process as they do in the rest of the initiatives that are described next.

MDA and its related technologies provide designers and developers with a set of tools for creating and manipulating models for a variety of purposes. The next section describes how all these concepts have been applied to User Interfaces.

[3] http://en.wikipedia.org/wiki/Model-driven_development

3.2 Model-Driven Engineering of User Interfaces

This section introduces the model-driven approaches used in the development of user interfaces. The section briefly describes the Cameleon Reference Framework and its different levels of abstraction. Then, an overview of the UsiXML language is introduced.

3.2.1 The Cameleon Reference Framework

Cameleon [18] is an unifying reference framework that "characterizes the models, the methods, and the process involved for developing user interfaces for multiple contexts of use, or so-called multi-target user interfaces". Here, a *context of use* is decomposed into three facets: the *user*, the computing *platform* including specific devices, and the complete *environment* in which the user is carrying out interactive tasks with the platforms specified. When variations of one or more of these facets (<user, platform, environment>) appear, they are referred as a change of the context of use that should or could be reflected in some way in the user interface.

According to [18], the Cameleon reference framework can be summarized with the following assertions (figure 3.11):

1. It clarifies what are the models used (e.g., task, concept, presentation, dialogue, user, platform, environment, ...), when these models are used (e.g. at design-time vs. run-time), and how they are used (i.e. at the four levels of concern: task and concepts, abstract UI, concrete UI and final UI) and according to which process.
2. It allows designers to use either a *forward engineering* approach from the highest level of abstraction of the framework (i.e. Tasks and Concepts) to the lowest level (i.e. the Final UI), a *Reverse engineering* approach, that takes the inverse path, or a *Bidirectional engineering* or re-engineering approach, which is a combination of both forward and reverse engineering.
3. It allows to use different entry points to the process. This means that the process can be initiated at any point of the framework and not only at the top (like in a full top-down

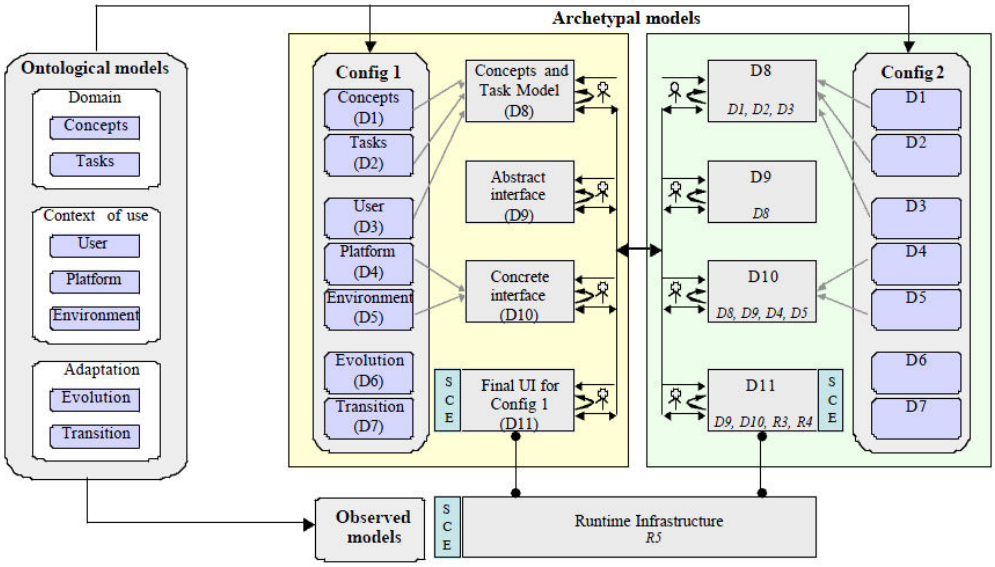


Figure 3.11: The Cameleon Reference Framework ([18])

approach) or at the bottom (like in a full bottom-up approach).

The transformations used between models of the UI receive special nouns regarding the levels of abstraction of the source(s) and target(s) model(s):

Reification is the transformation where the source model(s) has a higher level of abstraction than the target model(s). It is usually related to a top-down transformation process. Is the normal kind of transformations used in a forward engineering process, for instance, in the generation of the source code by transformation of a concrete UI model.

Abstraction is the transformation where the source model(s) has a lower level of abstraction than the target model(s). It is usually related to a bottom-up transformation process, classically used for reverse engineering purposes.

Reflexion is a transformation that updates the model itself, and in consequence, keeping the level of abstraction.

Translation is the transformation that produces a target model of the same level of abstrac-

tion as the source model. It is specially useful for migrating the UI from one context of use to another.

3.2.2 Levels of Abstraction

The Cameleon Reference Framework is composed of four main levels of abstraction. These leves are the Task and Concepts level, Abstraction level, Concrete level, and Final UI level, each of them representing one different aspect of the user interface. Figure 3.12 shows the different levels of the framework. Each of the layers represents a model.

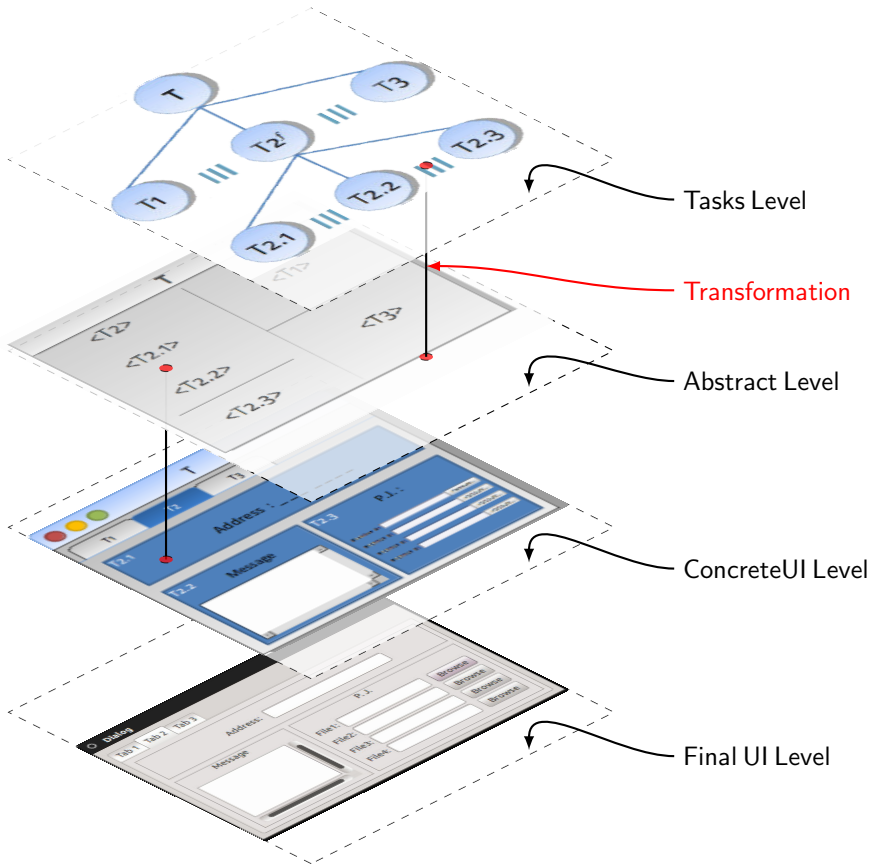


Figure 3.12: Cameleon levels from final user interface to tasks level. Two transformations are drawn with straight lines. The source and the target of the transformations are outlined with circles.

Tasks and Concepts model. It represents the tasks that the user can perform on the UI. The tasks manipulate the concepts, also represented at this level. Figure 3.13 shows a Task model in CTT [115] for a help system. The root task is an abstract task that is decomposed into four sub-tasks of different types. The first task is a *User task* in which the user does not interact with the system. The second task is an *Interactive task* in which the user request some information from the help system. The third and fourth tasks are *Application tasks* in which the system computes the required information (third) and provides it back to the user (fourth). Note that these tasks are platform independent. In this sense, Task-models can be considered as a PIM model from the MDA perspective.

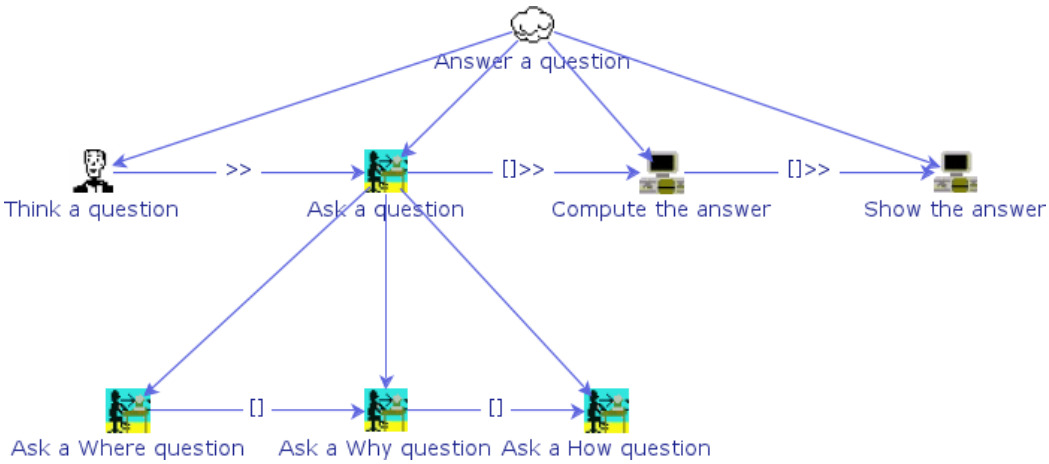


Figure 3.13: Example of task model. Modelisation of a help system that supports three different types of questions that the user can ask.

Abstract UI. It groups the tasks in a platform independent way, i.e., without taking into account the details of the platform or platforms where the user interface will be running after being generated. For this reason, AUI models can be also considered as PIM models with regard to MDA. Figure 3.14 shows a proposition of Abstract UI model for the help system modelled in the previous task model. All the interaction takes place in the same space, called here AUI Container. Two different units are defined inside such space.

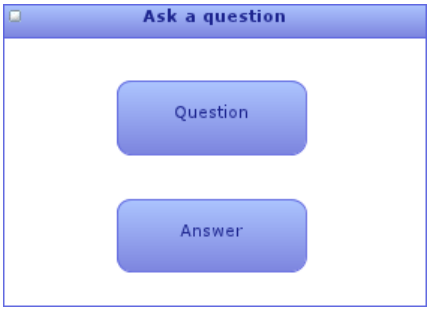


Figure 3.14: Abstract UI model (AUI).

Concrete UI. It structures the elements of the abstract user interface into platform dependent elements. For instance, for Graphical User Interfaces or GUIs, the Concrete UI defines what widgets are necessary for each element of the abstract user interface. Figure 3.15 shows a proposition of a Concrete UI model for the previous a AUI model. In this case, the space of interaction becomes a window whereas the two AUI units have been transformed into a TextField and a Label respectively.

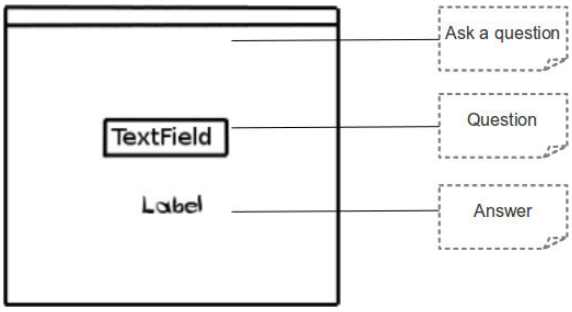


Figure 3.15: Concrete model (CUI).

Final UI. It represents the source code implementing in a specific programming language the widgets (in the case of GUIs) of the previous level. In the example shown in figure 3.16, the implementation of the CUI has been done in a Java framework.

The source code of the UI, i.e., the Final UI, is obtained by iterative transformations of different UI models, generally from top to bottom. For instance, to obtain the final UI shown

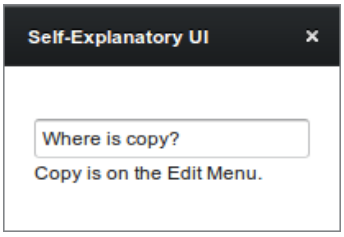


Figure 3.16: Final UI (FUI).

in figure 3.16, the task model has been transformed into the AUI model, which is then transformed in turn into the CUI model, from which the Final UI is finally derived. Figure 3.12 summarizes this process graphically.

The Cameleon Reference Framework has been reviewed, extended, and improved, in the UsiXML language, which will be discussed in the next section.

3.2.3 The UsiXML Language

UsiXML can be considered as a natural evolution of the Cameleon reference framework. The UsiXML language preserves the four levels of abstraction, from the Tasks level to the FUI, reviewing their models and meta-models, and it extends the Cameleon Reference Framework with new models that add new functionality to the UsiXML language.The meta-models of our research and the UsiXML meta-models have been reciprocally enriched during their mutual development. This section will briefly present the meta-models of the UsiXML language that are not covered by the Cameleon Reference Framework, that are relevant in the context of our research.

The UsiXML language is defined (slightly adapted from [149]) as:

> An innovative model driven language that attempts to improve the UI design, for the benefit of both industrial end-users actors in term of productivity, reusability, usability, and accessibility, by supporting the μ7 concept: multi-device, multi-user, multi-culturality/linguality, multi-organisation, multicontext, multi-modality and

> multi-platform.

The μ7 concept is supported through a set of models and meta-models that composes the UsiXML language. The meta-models and models that we have used in our research are largely inspired by the UsiXML meta-models. Here is a brief description of the most relevant UsiXML meta-models for our research[4]:

Task Meta-Model It aims to define the tasks that the user can perform in the system. The Task Meta-Model represents the same concepts as the one previously described in the Cameleon Reference Framework.

Domain Meta-Model It consists of a description of the classes of objects manipulated by a user while interacting with the system. It specifies the main concepts of a User Interface by identifying the relationships among all the entities within the scope of such User Interface, their attributes and the methods encapsulated within the entities.

Abstract UI Meta-Model is an expression of the UI in terms of interaction spaces (or presentation units), independently of which interactors are available and even independently of the modality of interaction. It represents the same information found in the AUI model from the Cameleon Reference Framework.

Concrete UI Meta-Model It is an expression of the UI in terms of "concrete interaction units", that depend on the type of platform and media available. It represents the same information found in the CUI model from the Cameleon Reference Framework.

QOC Meta-Model This meta-model supports design rationale based on the QOC (Questions, Options, Criteria) notation [84]. It supports the exploration of options during design processes.

Transformation Meta-Model The aim of this meta-model is to define how transformations are composed in UsiXML.

[4]For more information about the UsiXML meta-models, please visit `http://www.usixml.org/`.

Quality Meta-Model It aims at providing means for integrating quality criteria into the design process of the user interface. This meta-model is a contribution of this research to the UsiXML language. It provides end users with design rationale explanations based on design decisions.

The quality meta-model is entirely described in chapter 5. Quality models are useful because they can help to support the design rationale axis identified in the problem space. To better understand the role of quality models and how they characterize quality, the next section overviews the most relevant quality models of the literature.

3.3 Quality Models

This section reviews the major quality models that have been used in software engineering as well as some ergonomic guides that are relevant for HCI. This is necessary in order to understand what a quality model is, what the role and contributions of these different quality models are, as well as the importance that ergonomic criteria have in user interfaces. Quality is important for providing users with design rationale explanations.

3.3.1 Quality Model Definition

The ISO 14598-1 Standard for Information technology and Software product evaluation ([62]), defines the term *Quality Model* as:

> "The set of characteristics and the relationships between them which provides the basis for specifying requirements and evaluating quality."

Quality Model definitions are usually generalist. The previous definition does not provide any information about how to determine a characteristic, how to characterize a relationship between characteristics or how to specify the requirements that are measured in a quality evaluation. The next sections will cover how these questions have been addressed by different quality models. These quality models are summarized in figure 3.17.

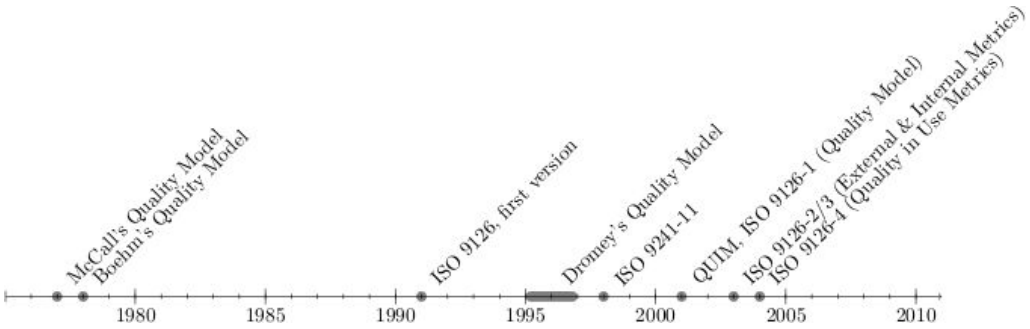


Figure 3.17: Relevant works on quality in their years of publication.

3.3.1.1 McCall's Software Quality Model

McCall's hierarchical quality model [88] is one of the earliest. The model aims to "bridge the gap between users and developers by focusing on a number of software quality factors that reflect both the users' views and the developers' priorities".[88]. It also aims to "provide a complete software quality picture" ([67]). To this end the model organizes the product quality into two views: the external view for the client and the internal view for the developers. These views are decomposed in the model (figure 3.18) as follows:

Factors : They describe the external view of the software, as viewed by the users.

Criteria : They describe the internal view of the software, as seen by the developers.

Metrics : They are defined and used to provide a scale and method for measurement.

The external view (the users' view) consists of 11 quality factors, while 23 quality criteria describe the internal view of the software (developer's view).

The idea behind McCall's Quality Model is that the quality factors synthesized should provide a complete software quality picture. McCall's quality model makes a first step forward by subdividing and categorising Factors, Criteria and Metrics.

3.3.1.2 Boehm's Quality Model

Boehm's quality model ([12]) is decomposed in a hierarchical way as McCall's model does, but contrary to this, the top of the model addresses the end-users' concerns while the bottom

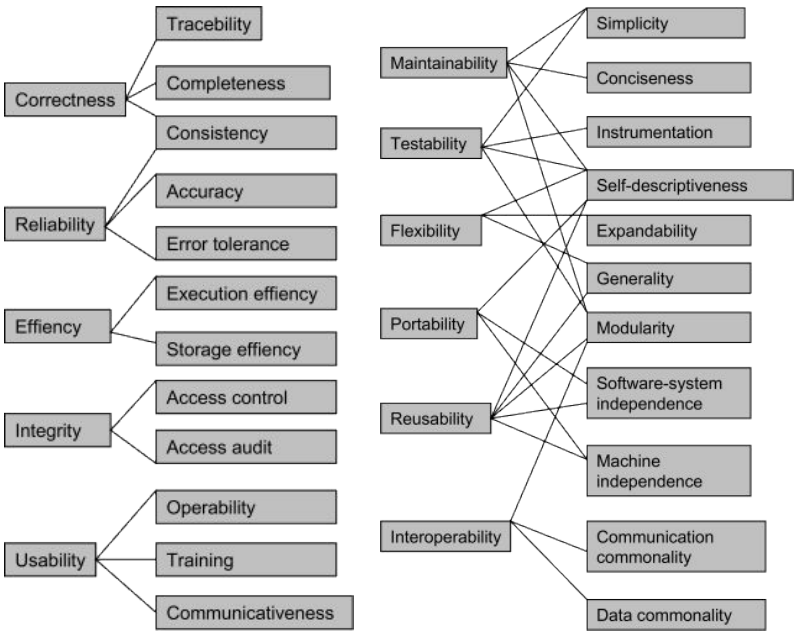


Figure 3.18: McCall's quality model: 11 quality factors are decomposed into 23 quality criteria.

addresses the designers' perspective. This is mainly due to the emergence of the user's perspective of quality. However, even if the top of the model addresses end users' concerns, the model is quite far from the real perspective of the user as stated by [5]: "this interest wanes when one reads Boehm's definition of the characteristics of software quality. Except for General Utility and As-is Utility, all definitions begin with *Code possesses the characteristic [...]*."

The General Utility characteristic (see figure 3.19) aforementioned in the citation is the major high-level characteristic of quality in Boehm's model (users' perspective at the top of the model) because, in words of [117], "a software system must be useful to be considered a quality system".

Boehm decomposes the overall quality into high-level characteristics, intermediate-level characteristics and primitive (or lowest-level) characteristics (figure 3.19).

The main difference between Boehm's and McCall's quality models, is that McCall focuses on precise measurements of high level, while Boehm presents a wider range of primary features. The high-level characteristics of Boehm's model (like General Utility and As-is are too

generic and imprecise to be useful for defining verifiable requirements. However, some authors declare that "Like the McCall model, this model is mostly useful for a bottom to top approach to software quality" [5], i.e. it can effectively be used to define measures of software quality.

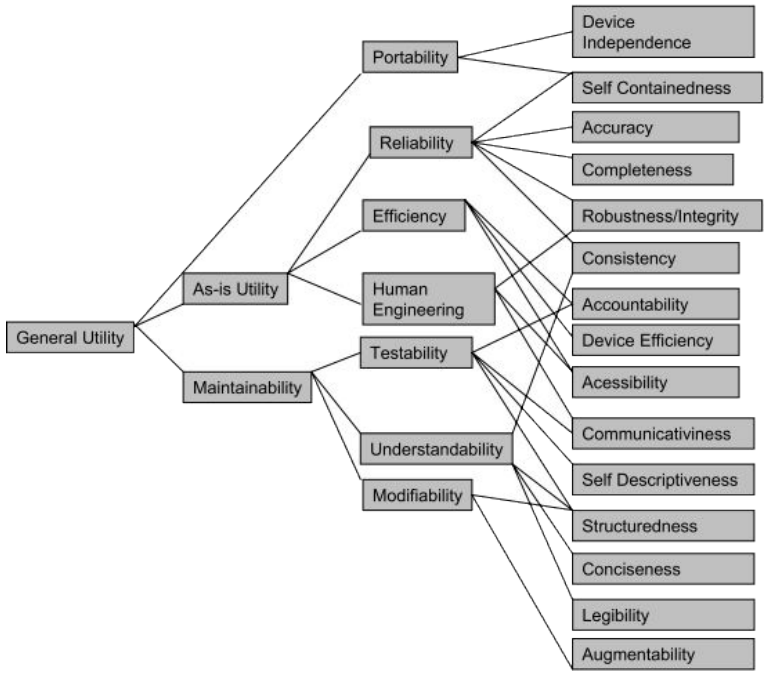


Figure 3.19: Boehm's quality model.

3.3.1.3 Dromey's Quality Model

Dromey's model ([33, 34]) takes a different approach to software quality. Dromey states that quality evaluation differs for each product and that "a more dynamic idea for modeling the process is needed to be wide enough to apply for different systems" ([33]). For Dromey, a quality model should be based upon the product perspective of quality: "What must be recognized in any attempt to build a quality model is that software does not directly manifest quality attributes. Instead it exhibits product characteristics that imply or contribute to quality attributes". Dromey suggested three prototypes concerning quality, which are the implemen-

tation quality model, the requirements quality model, and the design quality model. Figure 3.20 shows the implementation quality model.

Dromey's model is focused on the relationship between quality attributes and sub-attributes, trying to connect properties of software with software quality attributes.

The "Software product" is defined by the root of the model named *Implementation*. The "Software product" is subdivided into four "Product properties", from Correctness to Descriptive. Each of these "Product properties" is decomposed into different "Quality attributes".

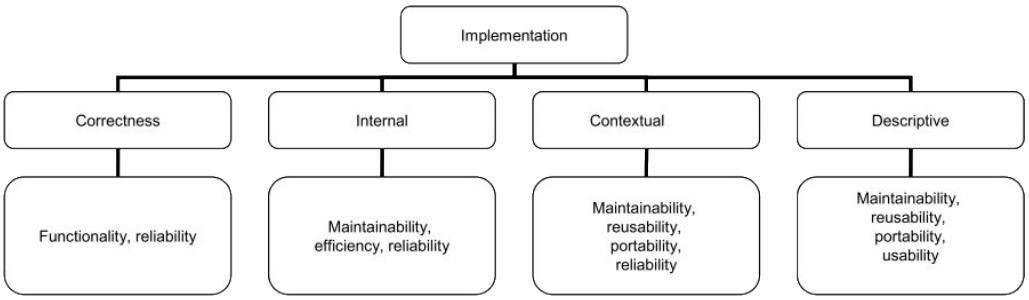


Figure 3.20: Dromey's implementation quality model.

As Dromey's model relies exclusively on software properties, some authors state that the model is not suitable for addressing the users' needs. For instance, [5] states that "this model is rather unwieldy to specify user quality needs".

3.3.2 ISO Standards

The International Organization for Standardization (ISO) has actively participated in the development of quality standards by presenting several propositions for addressing quality in different areas. In this report we mainly consider those standards that are useful for HCI, in particular those dedicated to usability.

In 1991, the ISO published the *Software Product Evaluation - Quality Characteristics and Guidelines for Their Use* (ISO 9126 [111]), which represents the first international consensus on the terminology for the quality characteristics for software product evaluation. Different approaches or *perspectives* of the concept of quality start to appear through these standards,

and the concept of usability evolves accordingly. The first of these kinds of perspectives appears in 1996 under the name of *external quality*, defined in the first part of the *ISO/IEC DIS 14598 Information Technology - Evaluation of Software Products* as "the extent to which a product satisfies stated and implied needs when used under specified conditions".

In 2001, a new standard is published under the name *ISO/IEC 9126 Software engineering - Product quality*. This standard has become one of the most important quality standards in software engineering. It is divided into four parts: one International Standard (IS) and three Technical Reports (TRs) that have been published in the next years:

- Quality Model (ISO IS 9126-1, 2001)
- External Metrics (ISO TR 9126-2, 2003)
- Internal Metrics (ISO TR 9126-3, 2003)
- Quality in Use Metrics (ISO TR 9126-4)

The quality model proposed in this standard is based on the McCall's hierarchical model, and it handles the same notions of Factors, Criteria and Metrics. In this first part, ISO provides a new definition for *usability* "as a product measure", and *quality in use* as "an outcome of interaction". ISO also redefines the concept of *External quality* as:

> the totality of characteristics of the software product from an external view. It is the quality when the software is executed, which is typically measured and evaluated while testing in a simulated environment with simulated data using external metrics. During testing, most faults should be discovered and eliminated. However, some faults may still remain after testing. As it is difficult to correct the software architecture or other fundamental design aspects of the software, the fundamental design remains unchanged throughout the testing. (ISO/IEC, 2001a)

On the other hand, the third technical report on *Internal Quality* describes this concept as:

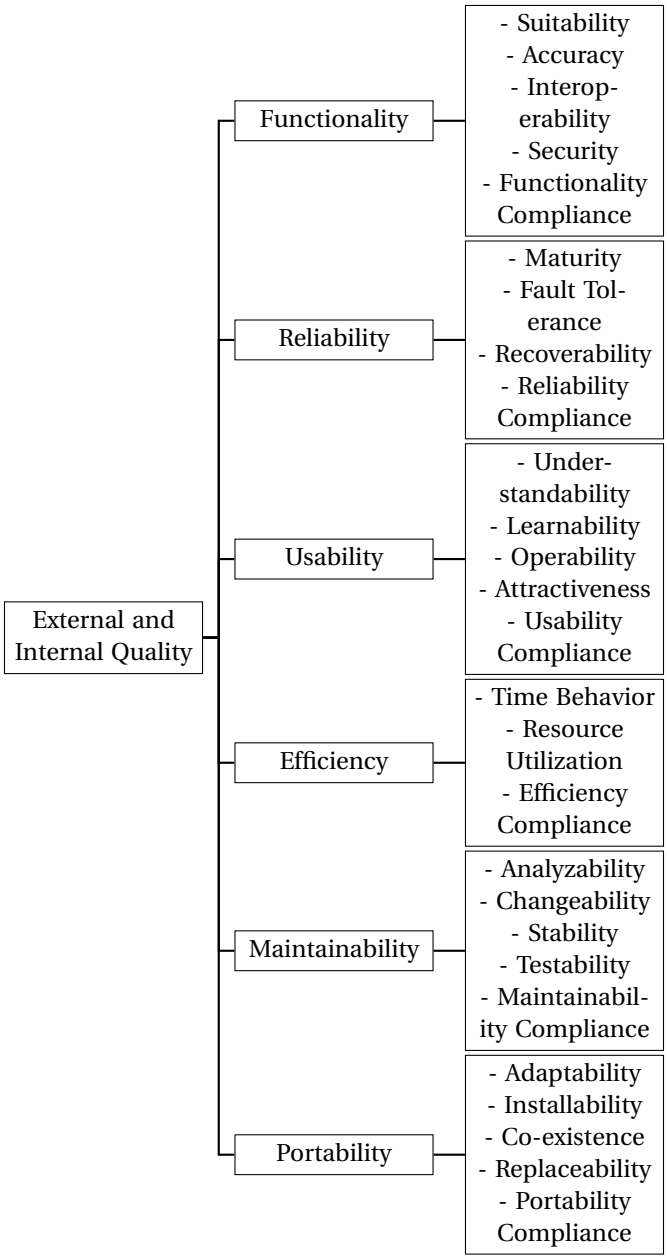


Figure 3.21: ISO 9126 quality model for external and internal quality.

> The totality of characteristics of the software product from an internal view. Internal quality is measured and evaluated against the Internal Quality requirements. Details of software product quality can be improved during code implementation, reviewing and testing, but the fundamental nature of the software product quality represented by the Internal Quality remains unchanged unless redesigned. (ISO/IEC, 2001a)

Finally, *Quality in Use* is redefined in the last technical report as:

> the user's view of the quality of the software product when it is used in a specific environment and a specific context of use. It measures the extent to which users can achieve their goals in a particular environment, rather than measuring the properties of the software itself. (ISO/IEC, 2001a)

In parallel with the evolution of the different ISO quality standards, some authors proposed alternative solutions to unify the concept of quality. This is the case of the QUIM model described next.

3.3.3 QUIM Model

The QUIM model [130] is a framework for quantifying usability metrics in software quality models. Seffah et al. encompass most of the usability works in the aforementioned QUIM or *Quality in Use Integrated Map*.

QUIM defines quality in use as:

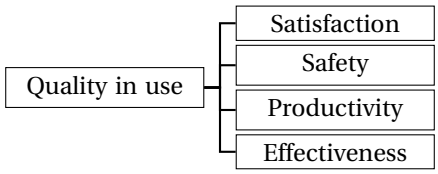


Figure 3.22: ISO 9126 quality model for quality in use (characteristics).

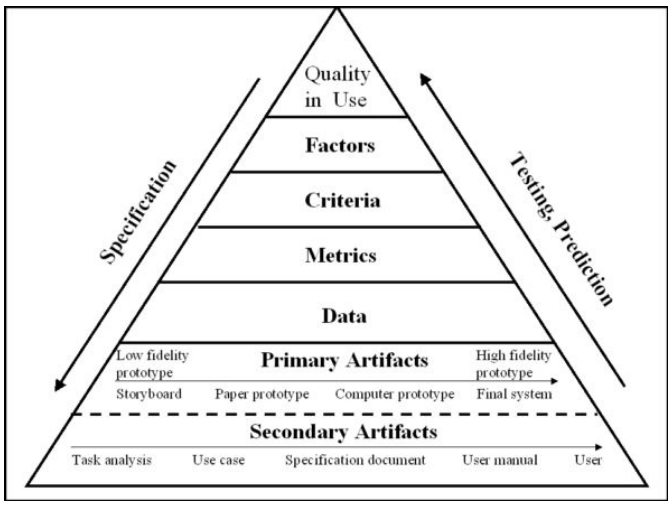


Figure 3.23: QUIM Structure and Usages.

> The end user perspective of software quality.

QUIM is an integrated framework for measuring and specifying quality in use models. QUIM establishes what factors, criteria and metrics should be developed, and what data should be gathered to calculate these metrics.

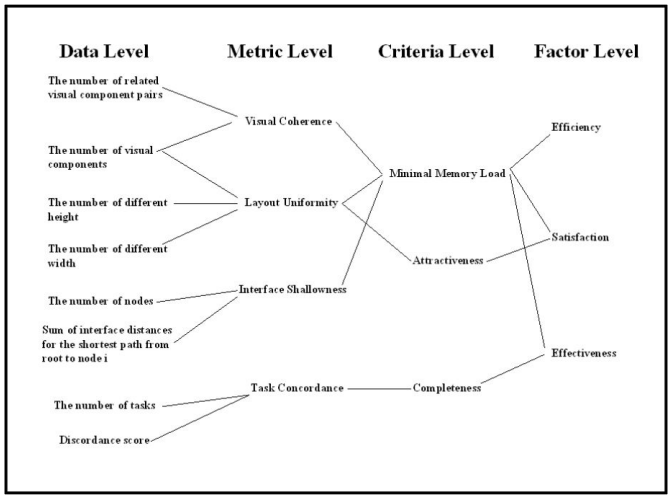


Figure 3.24: Example of components relationships.

In QUIM, the analysis of the authors of the existing models conducted them to the definition and validation of 7 factors, 12 attributes and more than 100 metrics that are integrated into QUIM. The Quality in Use Integrated Map uses a *Graphical Dynamic Quality Assessment* (GDQA) model to analyse interaction of these components into a systematic structure.

Keeping the same idea of unification shown in QUIM, Auvo Finne proposes his own quality meta-model described in the next section.

3.3.4 Finne's Quality Meta-Model for Information Systems

Finne's quality meta-model is depicted in figure 3.25 (adapted from [40]). The numbering and the arrows in the figure indicate the relative order in which the major model elements first come into focus during quality modelling. The process starts with selecting informants and attributes, and it ends by creating quality metrics. The whole model can be divided into six main parts:

1. actor and informants
2. the attribute set
3. the attribute model
4. metrics

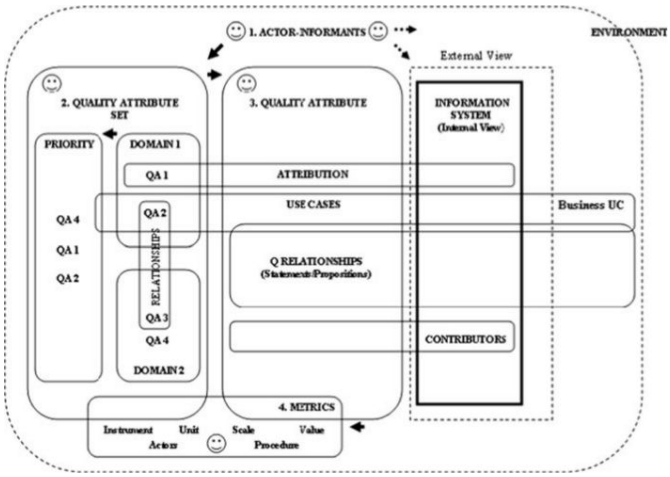


Figure 3.25: Finne's quality meta-model extracted from [40].

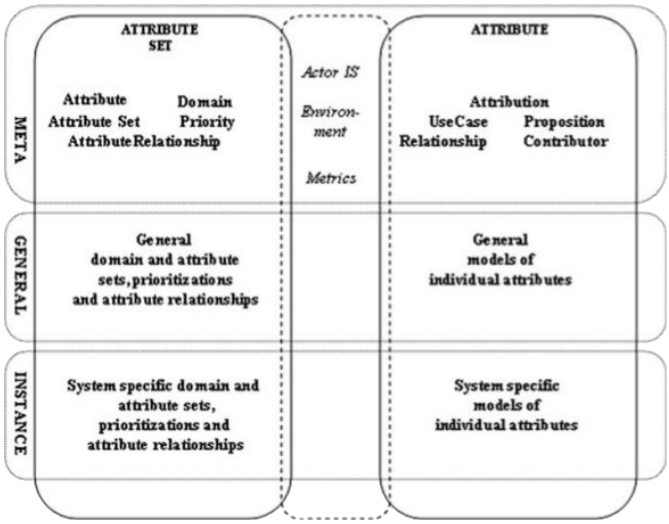


Figure 3.26: Levels of abstraction in quality modelling according to [40].

5. the information system
6. the environment

According to [40], the meta-model is characterized by its "three-level" nature which in words of the author (figure 3.26), "this refers to levels of abstraction needed in quality modeling. Discussion of the meaning of quality, attribute, collection, domain, attribute model, level of modeling, etc. belongs to the highest level". The lowest level is called the instance level and it makes reference to "all system- and project-specific considerations and descriptions that can be found at this level".

The previous meta-model can be applied to general information systems. In the case of user interfaces, we need concrete quality criteria that we can directly apply to UIs in order to provide explanations about the design rationale. For this reason, quality guides related to ergonomic criteria becomes specially relevant in the context of this research. These guides on ergonomic criteria are the subject of the next section.

3.3.5 Ergonomic Guides

This section briefly describes two important ergonomic guides for the ergonomic criteria in user interfaces.

3.3.5.1 Bastien and Scapin

In 1993, Bastien and Scapin presented a technical report entitled Ergonomic Criteria for the Evaluation of Human-Computer Interfaces [7]. The technical report presents first a brief summary of the research conducted towards the design of ergonomic criteria for the evaluation of human computer interfaces, and then, the full description of the most recent set of criteria. The summary outlines the context in which the criteria were developed, the goal of the criteria approach, the experiments conducted, and the results obtained.

The set of ergonomic criteria that resulted from this work consists of a list of 18 elementary criteria (including the 8 main criteria). The criteria are presented along with their definitions, rationales, examples of guidelines, and comments setting out the distinctions between some of them.

The main criteria are the following:

1. Guidance subdivided into Prompting, Grouping (either by Location or Format), Immediate Feedback, and Legibility.
2. Workload consisting of Brevity (subdivided itself into Concision and Minimal Actions) and Information Density.
3. Explicit Control which is composed of the subcriteria Explicit User Action and User Control.
4. Adaptability subdivided into Flexibility and User Experience.
5. Error Management composed of Error Protection, Quality of Error Messages, and Error Correction.
6. Consistency that refers to the way the user interface design choices are maintained in similar contexts.

7. Significance of Codes that qualifies the relationship between a term and its reference.
8. Compatibility referring to the match between, on the one hand, the task characteristics and users' characteristics such as memory or skills, and on the other hand, the organisation of the output, input and dialogue.

These criteria have also been used in different other guides. One of them is the Ergonomic Guide by Jean Vanderdonckt which is presented in the next section.

3.3.5.2 Vanderdonkt's Ergonomic Guide

Ergonomic rules are designed to enforce one or more criteria in the ergonomic design of user interfaces. The Vanderdonckt's Ergonomic guide [152] is composed of more than 3700 rules described along eight selected ergonomic criteria: *Compatibility*, *Consistency*, *Workload*, *Adaptability*, *the Control of the dialogue*, *Representativeness*, *Guidance* and *Management of errors*.

Vanderdonckt characterizes each criterion by a name, a definition, a goal, and a hierarchical decomposition in basic criteria according to linguistic levels also defined in his work.

This guide is probably the largest recompilation of ergonomic criteria until today, and it clearly shows the big effort that has been put in bringing quality to user interfaces.

3.4 Synthesis

In this chapter we have briefly reviewed the foundations that are necessary to understand our research. We have firstly described the model-related initiatives as well as their terminology and underlying concepts. We have then reviewed how the model-driven approach is currently applied to the domain of HCI, describing the Cameleon Reference Framework, its models for each level of abstraction, understanding how user interfaces are generated from models with this approach. We have then briefly review the UsiXML language as well, the natural evolution of the Cameleon Reference Framework.

Being able to qualify the quality of user interfaces is important to answer questions about

the design rationale. To this end, the chapter reviews how different authors have qualify the quality in different aspects. The chapther then stresses the important of ergonomic criteria for user interfaces, showing two well-known ergonomic guides.

Despite all this effort in producing high quality user interfaces, users still find problems in the interaction. Based on these foundations and the related work presented in chapter 2, we have now all the necessary elements to propose a solution for supporting users through models, which is the subject of the next chapter.

4

Self-Explanatory User Interfaces

> *Just because we don't understand doesn't mean that the explanation doesn't exist.*
>
> Madeleine L'Engle,

RELATED PUBLICATIONS

1. García Frey, A., Calvary, G., and Dupuy-Chessa, S. Users need your models! exploiting design models for explanations. In *Proceedings of HCI 2012, Human Computer Interaction, People and Computers XXVI, The 26th BCS HCI Group conference (Birmingham, UK)* (2012)
2. García Frey, A., Calvary, G., and Dupuy-Chessa, S. Xplain: an editor for building self-explanatory user interfaces by model-driven engineering. In *Proceedings of the second ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2010)* (2010), ACM Press, pp. 41–46
3. García Frey, A., Calvary, G., and Dupuy-Chessa, S. Self-explanatory user interfaces by model-driven engineering. In *Proceedings of the second ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2010)* (2010), ACM Press, pp. 341–344

As argued in previous chapters, the approach used in our research to support users at runtime is Model-Driven Engineering (MDE). Model-Driven UIs are able to explore the UI models at runtime, extracting the necessary information to support users. In this thesis, we try to unify the extraction and exploitation of explanations from design models through different contributions.

Firstly, the chapter describes the concept of *Gulf of Quality*, an extension of Norman's Theory of Action. The *Gulf of Quality* couples the perception of designers and users under the same framework. This extension sets the basis for the hypothesis of this thesis presented in the introduction, that allows us to state that the models used by designers at design time are useful for supporting end users at runtime.

After this and based on this hypothesis, the chapter describes the design principles for building model-driven help systems. They are described through a four steps approach. These design principles explain a method for developing self-explanatory user interfaces from a model-based perspective.

Later, the chapter describes a set of *explanation strategies* that are used by the self-explanatory user interface to retrieve the necessary information from the underlying models of the UI, and compose the answer based on that information.

Finally, the chapter ends with a synthesis of the proposed solution.

4.1 Introduction

Self-Explanatory UIs were defined as user interfaces with the ability of providing end-users with information about the UI, in order to support the users at runtime. As an example, consider the image shown in figure 4.1.

In this screenshot a message is displayed by request when the user asks information about the window in the background. The objective is to generate these answers automatically at runtime.

In the example of the figure, the user is requesting information from the UI through a

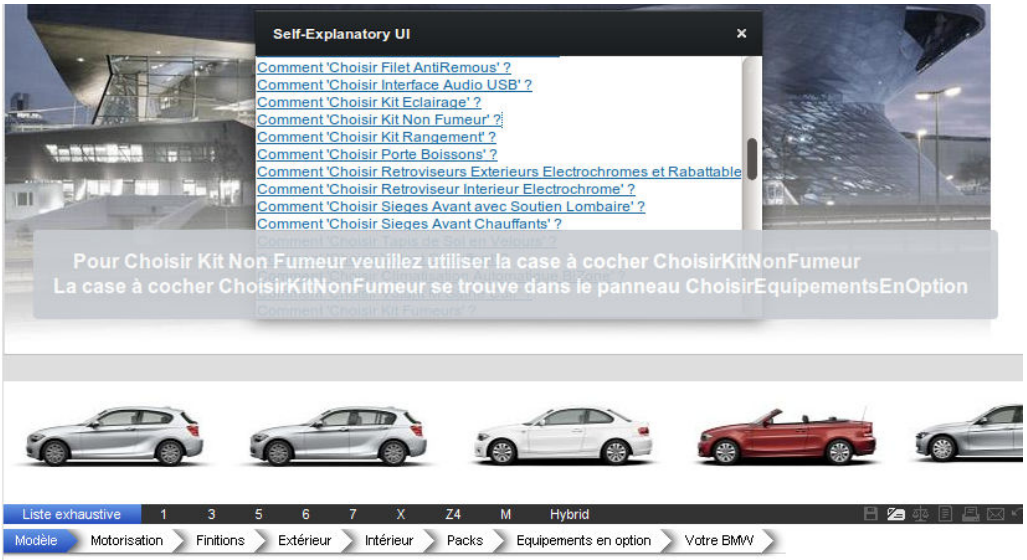


Figure 4.1: A help message leads the user through the UI. English translation: "To select the Non-Smoker Kit use the 'Select Non-Smoker kit' CheckBox. The 'Select Non-Smoker kit' CheckBox is located in the 'Select Extra Equipment' panel."

dialogue that proposes questions to the user. Once the user clicks on the question he/she wants to ask, a help message is shown with the desired information.

Technically speaking, self-explanatory user interfaces can support users in different ways using many different technologies. In the chapter State of the Art we have presented several works from a diversity of computer science domains that address this problem from different perspectives. In our research, we chose to explore model-based approaches of user interfaces for several reasons. These reasons are discussed in chapter 2. To build self-explanatory user interfaces with models, we firstly propose to extend the Norman's Theory of Action with an extension called *Gulf of Quality*. The description of both is the goal of the next section.

4.2 Gulf of Quality

Inspired by the Isatine framework[1] [82], we reuse Norman's Theory of Action to define the hypothesis on which the concept of *Gulf of Quality* is based. Norman stated ([106]) that any action of the interaction between humans and computers consists of seven cyclic stages. These stages are categorized into two gulfs (figure 4.2) that designers must ideally overcome:

Gulf of Execution is the gulf getting from the intention to execution.

Gulf of Evaluation is the gulf involved in interpreting and evaluating the system response.

The Theory of Action relies on the hypothesis that end users elaborate mental models of the interactive systems, and that these models determine end users' behaviour during the interaction. We extend the theory to explicitly consider design models (figure 4.3) as follows.

When the designer of a UI interacts with the interface as a normal user does (figure 4.3), according to the Theory of Action he/she makes mental models that determine the interaction process. However, we claim that the designer's behaviour is also determined by other models related to the design process.

Examples of these models that may influence the designers's behaviour are task models and design rationale, classically expressed using notations such as QOC ([83]) or DRL ([73]).

Because designers of an interactive system understand the system they design, their models are supposed to be more complete and accurate than end users' mental models. This fact explains why designers don't need the same support as end users and why they don't find the same problems in the interaction. Moreover, some works identify design models as being key for understanding the UI, for example [133] stated that changing the platform of a UI leads to the reexamination of the initial designs. This fact sets the basis for our working hypothesis already presented in the introduction:

[1] ISATINE [82] is a multi-agent architecture that decomposes the adaptation of a user interface into steps that can be achieved by the user, the system and by other external stakeholders. The user can take control of the adaptation engine by explicitly selecting which adaptation rule to prefer from an ad- aptation rule pool in order to express the goal of the adaptation more explicitly but does not provide a mechanism to utilise multiple configuration techniques at run-time.

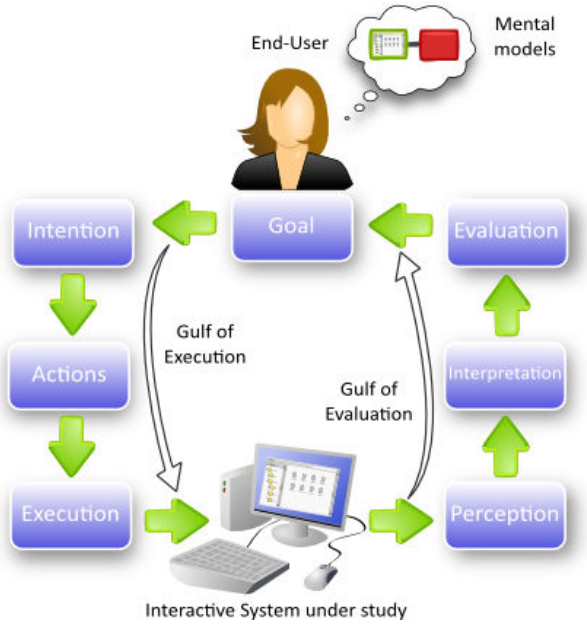


Figure 4.2: Norman's Theory of Action.

> Design models are suitable for supporting end users in the interaction process.

The immediate consequence is that design models can enrich end users' support, so they will better understand the UI. Therefore, they'll have less problems in the interaction.

To directly take into account design models for supporting purposes, we introduce the concept of *Gulf of Quality*.

We define *Gulf of Quality in interaction*, or simply *Gulf of Quality* as

> The distance between the design models the designers create at design time and the mental models the end users make at runtime while interacting with the system.

Figure 4.4 shows a graphical representation of the concept of *Gulf of Quality*. Note how the term "design models" considered in the previous definition has a different sense as in the Theory of Action. Norman denotes "design models" to the designer's mental model ([107], page 47), while we explicitly consider the design models used to develop and produce the

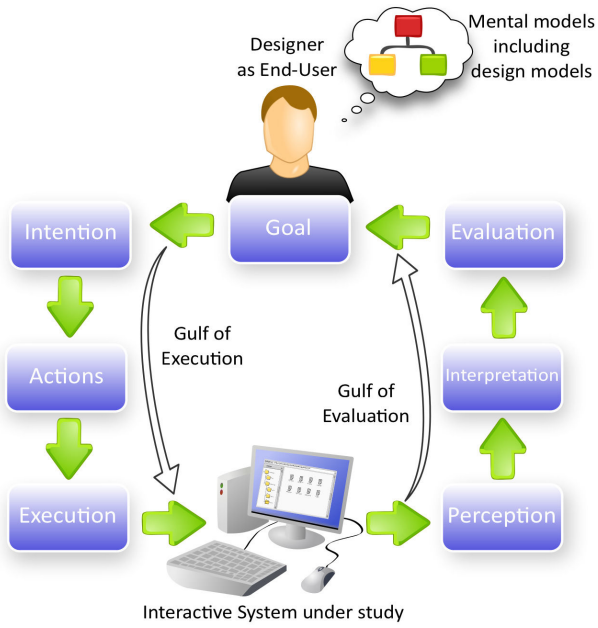


Figure 4.3: Design models influence the designer's behaviour in the interaction process.

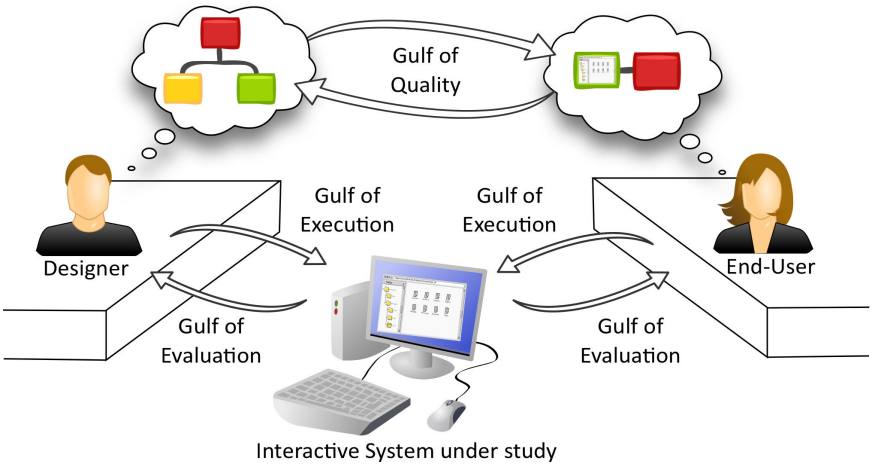


Figure 4.4: Gulf of Quality.

user interface from a model-based perspective.

Model-Driven approaches are suited for reducing the *Gulf of Quality* for several reasons. Firstly, these design models are explicitly defined by designers at design time. In fact, as shown in the chapter Foundations, models are used to directly generate the user interface. For instance, the Final UI in the Cameleon Reference Framework is generated from models such as the Task model or the Abstract UI model.

Secondly, a large effort has been put in model-driven engineering to keep models alive at runtime. And because these models are alive at runtime, they could be also used not only for generating the user interface, but also, if possible, to help users to better understand the user interface at runtime, providing them with explanations extracted from all these models.

The next section describes the design principles that are needed to reduce the *Gulf of Quality* by automatically extracting from design models the relevant information that is useful for supporting end users.

4.3 Design Principles

This section firstly establishes the necessary functionality that we consider for our help systems. Based on this functionality, we then present the design principles described through a four steps methodology. These principles are necessary to understand the conceptual solution as well as the architecture that implements this solution, which is presented later in section 4.4.

4.3.1 Help Systems Functionality

Help systems generated with our approach are responsible for:

- *Providing means for asking for support.* Designers must choose the way end users will ask for assistance so the system can understand the request. For instance, natural language dialogues or contextual help menus are valid for this purpose.
- *Computing the support the end user is asking for.* Once the question is understood by the system, its answer needs to be computed. For instance, if the end user asks how

to configure the recto-verso printing option, one possible answer the help system can compute is the necessary steps the end user needs to do to access the dialogue where this option is. Using our approach, the help system will query some design models to find the location of the recto-verso option and will compute the required steps that the end user needs to do to display it.

- *Presenting the computed support.* The computed answer must be provided to the end user in an understandable way. Natural language is a common option but designers can use any others with our approach, for instance, an animation of the mouse cursor that shows all the steps that are needed to configure the recto-verso option.

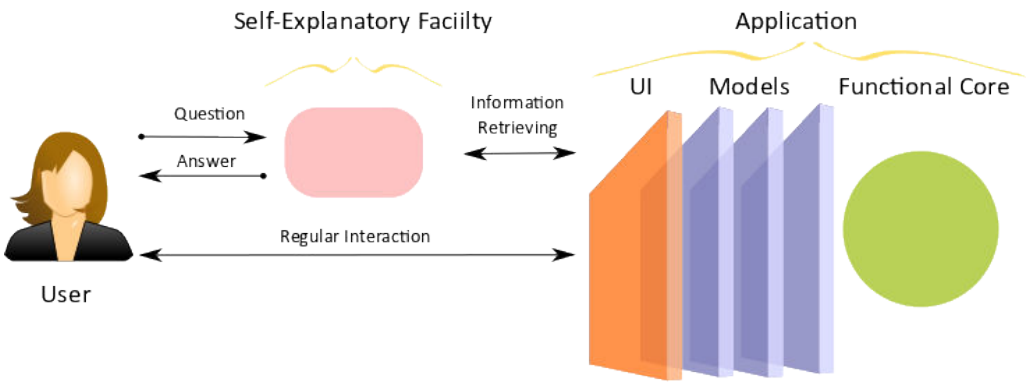


Figure 4.5: Explanation through a query paradigm.

We support users through an explanation query paradigm (for instance, [156, 69]), where users can obtain explanations to questions about the user interface. Figure 4.5 represents this approach. The user on the left side interacts with the application normally, which is represented through the *regular interaction* arrow. The represented application is a model-based application composed of models, but it could be of any other nature. When the user wants to request information about the user interface, he/she requests this information by asking a question to the help facility. This help facility, the self-explanatory facility in our case, receives the request and retrieves the information from the knowledge-base. This knowledge base is composed of models in our research. Once the necessary information has been lo-

cated and retrieved from the sources, the help facility computes the explanation and provides the answer back to the user.

As previously stated, our research proposes the use of models to support users at runtime. According to this and with the aim of supporting the previous functionality, the next section describes the design principles for building Model-Driven help systems supporting this functionality.

4.3.2 The Global Approach

According to the previous help systems functionality, the self-explanatory facilities generated with our approach are responsible for:

Generating the set of questions. In our research we consider those questions that the help system "knows" how to answer by inspecting the underlying models of the UI. For this reason, it is convenient to generate these questions as well. By doing this, designers can propose to users the questions for which the system knows an answer.

Generating answers. Once the user asks a question to the help system, the help system needs to compute an understandable explanation or answer. This is done through the following three steps:

Selecting the Explanation Strategy. In this phase the help system selects the explanation strategy that will be used for such a question. The explanation strategy is selected according to the type of the question, meaning that the explanation strategy responsible for answering "How" questions, will probably inspect different models, retrieving different information, from the explanation strategy responsible for "Why" questions.

Inspecting the models. Each explanation strategy will inspect one or more models to retrieve the elements that have been defined for each strategy. This elements will be used to compose the information that the user is requesting for.

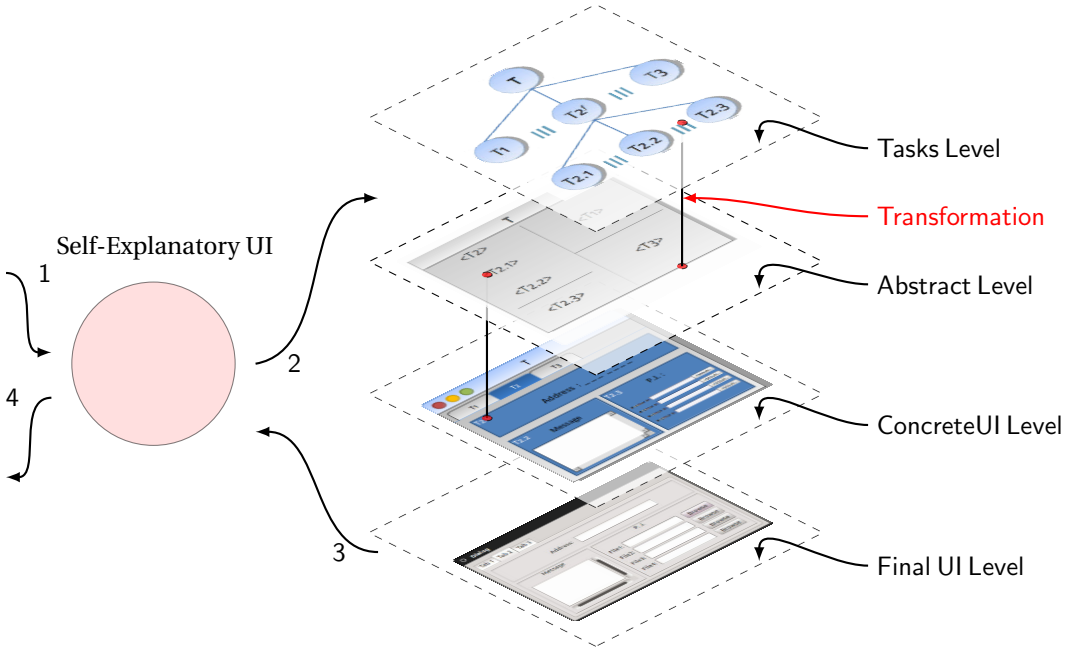


Figure 4.6: Global approach for Self-Explanatory help sytems.

Composing the answer. Once all the elements of the models have been retrieved, the answer can be composed and prepared to be presented.

Presenting the answer. The computed answer is provided to the user in an understandable way such as natural language.

Figure 4.6 describes the global approach graphically. First, a question is requested by the user (1). Each explanation covers a specific type of questions. Thus, the explanation strategy that corresponds to the question type inspects the models of the user interface at runtime (2). The explanation strategy retrieves all the elements from the underlying models (3), and it finally composes the desired answer that will be provided back to the user (4).

Consider the example described in figure 4.7. The user asks the question "How to select the external colour?" through the self-explanation facility (1). The self-explanatory UI inspects the models of the target application at runtime (2). The explanation strategy responsible for answering How questions, searches for the elements that are necessary to answer such

type of questions, it then retrieves these elements from the models, composing the answer which is presented later to the user (4).

With this global approach in mind, the next section describes the design principles for building model-based help systems.

4.3.3 Design principles

Design principles for explaining how to get end users' requests, how to extract explanations from design models according to these requests, and how to provide the extracted information back as support, are described through a four steps methodology. This methodology will ensure certain properties of these help systems, that are discussed later in the chapter. The four design principles cover respectively the four following questions:

1. how to build the UI of the help system
2. how to build the UI of the application
3. how to add support for computing help
4. how to weave both UIs into a one single user interface

The goal of these principles is to come up with a model-based self-explanatory facility that can be used to enrich the user interface of an application. The procedure of building such self-explanatory facility is summarized in figure 4.8. In the figure, the left side models are

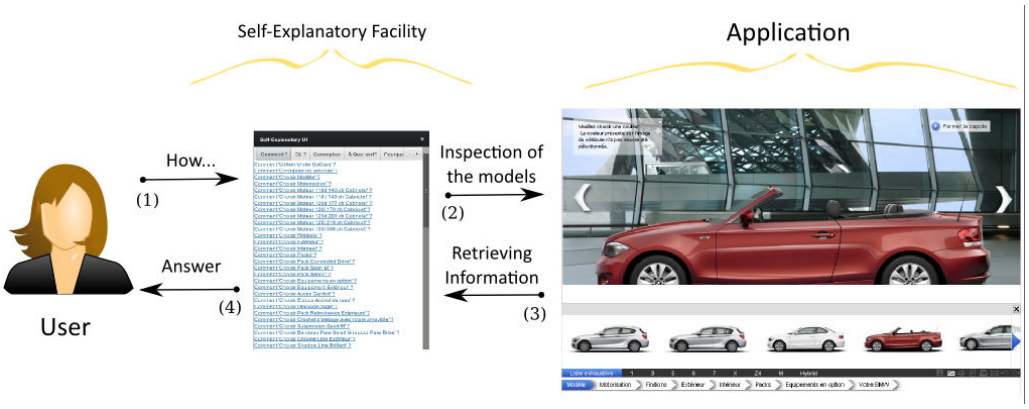


Figure 4.7: Example of the global approach.

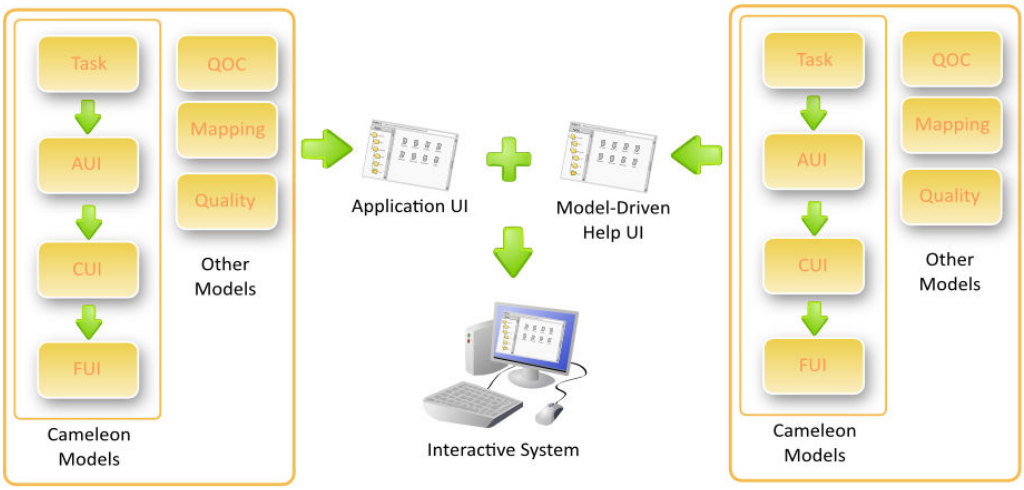


Figure 4.8: The design principles explain how to build a self-explanatory UI based on models that is able to answer questions about the UI of the Application.

those related to the user interface of the application, i.e., the models used for building such UI. Similarly, the models on the right side of the figure are those used for building the UI of the help system. The combination of both produces the full application with self-explanation ability.

The four principles for building such self-explanatory user interfaces are described in order in the following, illustrated through an example.

4.3.3.1 Building the UI of the Help System

First, we build the UI of the help facility according to a model-based approach as the one presented in the chapter Foundations, this is, defining the classical UI models in a first step, and applying then top-down transformations on these models to obtain the code of the user interface at the end of the process. In our research, we use the same four levels of abstraction of the Cameleon Reference Framework.

As an example, consider the construction of the help facility shown in figure 4.9. In this example, the task model at the top of the figure is transformed to a Abstract UI represented by blue boxes, which is in turn transformed into a Concrete UI represented with a mock-up.

This Concrete UI is then transformed into code producing the Final UI at the bottom of the figure. All the transformations that have been applied are represented with red arrows.

4.3.3.2 Building the UI of the application

In the second phase, the UI of the model-based application needs to be constructed following the same model-based approach that has been selected before. This is necessary to ensure certain of the properties that we are going to explain later in the chapter. From a model-based point of view, the models of the UI of the application and the models of the UI of the help system must conform to the same meta-models. For instance, if we are building the UI of the help system with the Cameleon Reference Framework models, the models of the UI of the application must be built using the same meta-models from the Cameleon Reference Framework.

Figure 4.9 summarizes this procedure graphically for the UI of an example desktop application.

Note that this approach does not set any restrictions on what models are needed to generate the UI. For those applications having non model-based UIs, reverse engineering techniques can be applied to obtain these models in a bottom-up transformation process from code to tasks ([81]).

4.3.3.3 Adding support for computing help

According to the hypothesis, the design models are suitable for supporting end users at runtime. In this phase, designers must add generic ways of computing explanations from these models in order to support the users at runtime. As previously shown in the chapter State of the Art, several works describe how to use specific models for this purpose. In particular, these works present specific solutions for specific types of questions. For instance, *Why* and *Why not* questions ([153] or *How* questions [114]

Our approach shows how to unify all these methods and how to use them together at runtime. Designers are free to exploit other models from other model-based approaches as

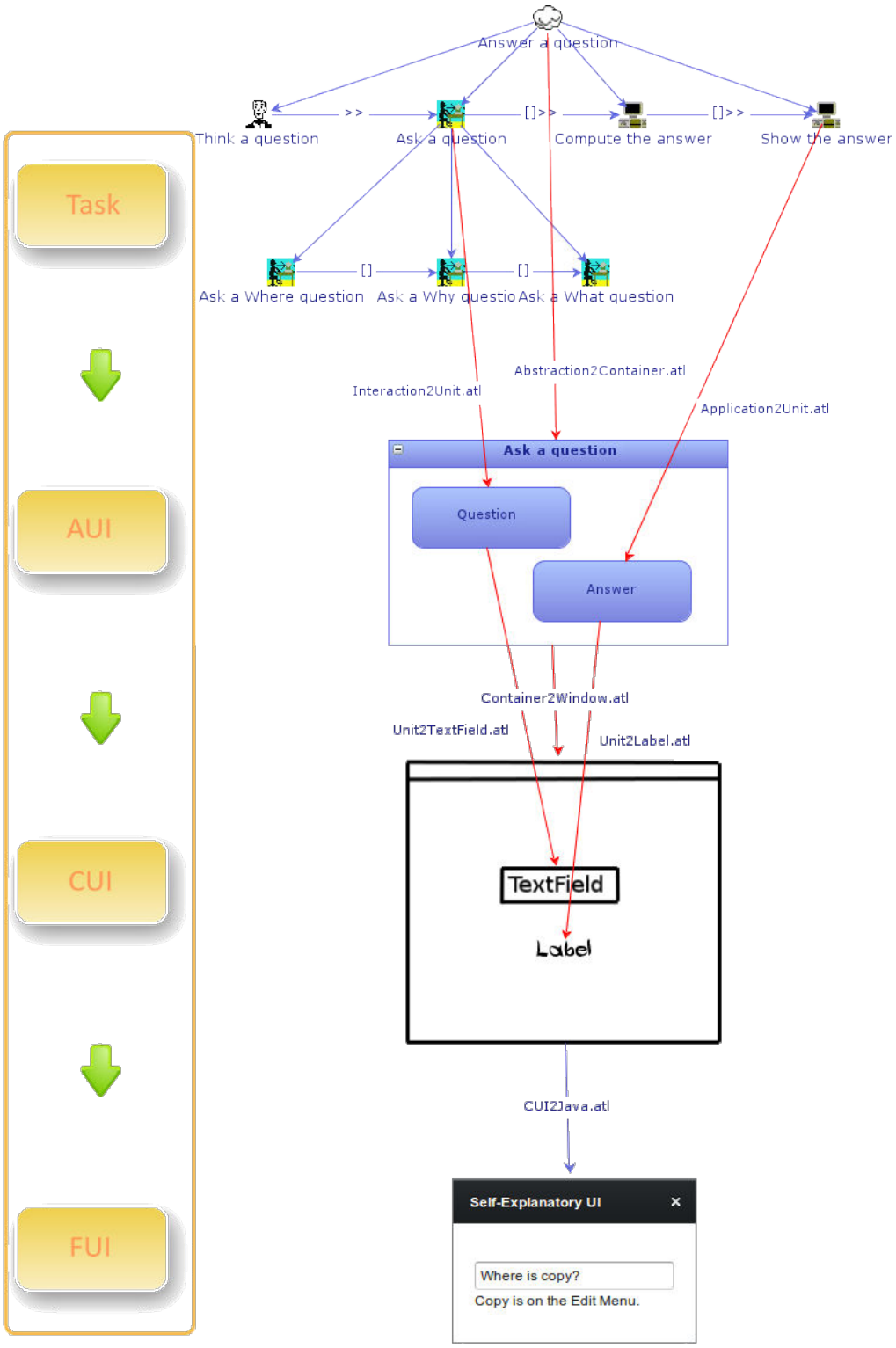


Figure 4.9: Building a model-based self-explanatory user interface.

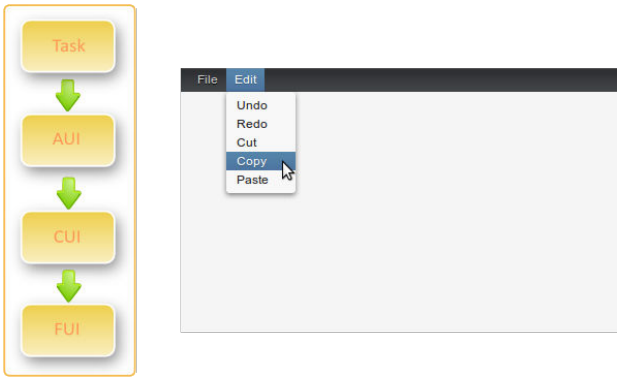


Figure 4.10: Building the UI of an example application according to the Cameleon Reference Framework. Left: Source models. Right: Excerpt of the Final UI.

there are no restrictions about what models to use. Note that these approaches are based on generic answers, i.e. designers do not need to write all the possible answers for all the possible "why this happens?" questions, but only the mechanism that computes the answer from the underlying models. In our research and according to the second hypothesis, we exploit the Cameleon Reference Framework models at runtime as the knowledge-base of the self-explanatory user interface. To this aim, we propose a set of *Explanation Strategies* that designers can use for computing different answers to different types of questions, all of them based on the models of this framework. Readers can find all the *Explanation Strategies* detailed later in this chapter.

4.3.3.4 Weaving the UIs

In the final step, designers can mix the help UI with the application UI at different levels. Models composition has been discussed by many authors in MDE (e.g. by [77]). As model composition is not the focus of our research, we briefly discuss some advantages and disadvantages of weaving the help UI with the application UI at different levels of abstraction. Figure 4.11 describes four possible ways of weaving both user interfaces. Each of the central arrows represent a different way of weaving or mixing both the UI of the application and the help system. Each colour represents a developing path from which a Final UI, composed with

both user interfaces, is generated.

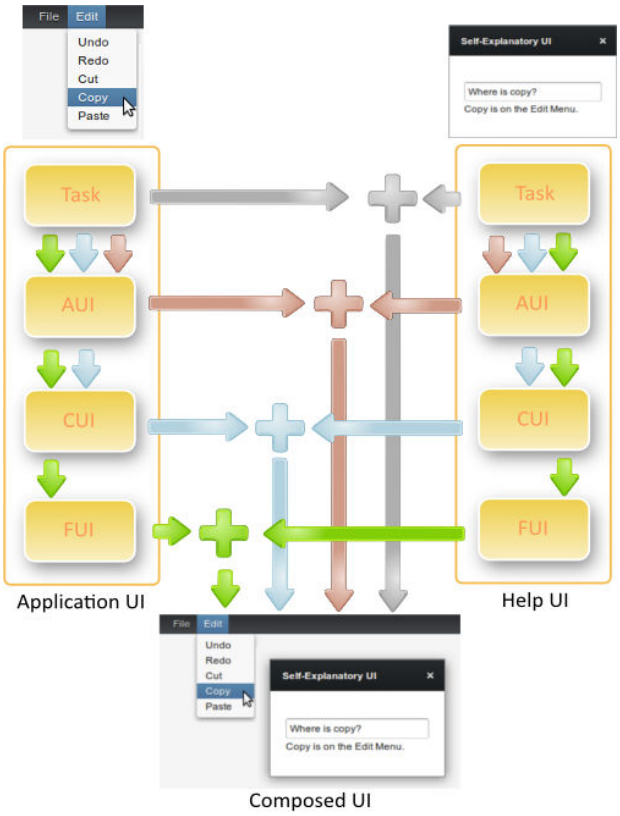


Figure 4.11: Generation of a model-based help system. Each column represents the models and transformations that produces the source code for the application UI (left) and the help UI (right). Different combinations for weaving both UIs are represented in the centre of the image.

For instance, the grey path represents a way of weaving both UIs at the task model level, i.e., weaving the task models of the UI of the application and the help UI into one single task model, and then transforming this task model into the correspondent Abstract UI, this one into a Concrete UI, which is finally transformed into the Final UI.

A second alternative path involves mixing both UIs at the Abstract UI level. In this case, the task models of the application and the help facility remains independent, and one AUI is composed containing the information from the two different AUI models. At this point, the procedure for generating the Final UI is again the same, transforming this AUI into a CUI, and

transforming the last one into the Final UI.

The third path is similar but mixing the UIs at the Concrete level. Thus, the UI of the application and the UI of the help system share both the CUI and Final UI models, but the task and abstract models from the previous levels remain independent.

A fourth possibility is to directly compose the UI at the Final UI level, mixing the code that has been generated by both transformation chains in an independent way.

Each of these alternatives present different advantages and disadvantages from the perspective of the help system. Weaving at higher levels of abstraction implies a decrease in the total number of models. For instance, if both UIs have been weaved at the task level, the total number of models in the final composed UI will be six, the two initial task models, one for the UI of the application and one for the UI of the help system, plus one task model resulting of the mixing of the previous models, plus the necessary AUI, CUI and FUI models. (Note that we are taking into account only those models directly related to the Cameleon Reference Framework, and not other external models such as quality models or design rationale notations).

The total number of models increases at the same time that we decrease the level at which the mixing is performed. Mixing at the AUI level results in 7 models following the previous reasoning, 8 models at the CUI level, and 9 models if the composition is done at the FUI level.

The implications of this for the help system are the following. First, as the help facility exploits design models at runtime, reducing the number of models will result in increasing the performance of the help system, because the number of models to explore is lower.

However, reducing the number of models by weaving at the more abstract levels has also the disadvantage of losing the information related to each UI. For instance, if the designer weave both UIs at the task level, the designer does not really know what elements of each model belongs to the help UI and what don't. One may need to make the distinction for many reasons such as for tuning the visual aspect of the help UI at the CUI level. In the case of weaving at the FUI level, the help models are completely separated from the application models, so customizing the UI of the help is easier because it only implies modifying the models of the

UI of the help before generating the code.

The choice of how to weave the UIs remains open for the designers, and our approach does not set any restrictions about how to do it. For example, the weaving of both UIs can be done even if only a limited set of models is available for one of the UIs. This is the case of those legacy applications where only the FUI of the application is available. Here, the self-explanatory facility can be modelled following the full top-down procedure involving the four levels of abstraction, and then, mix the resulting FUI of the self-explanatory facility with the FUI of the application.

According to the Global Approach early presented in section 4.3.2, the help systems built according to our approach and following these design principles need to define one explanation strategy for each type of question that wants to be supported. These explanation strategies are the subject of the next section.

4.4 Explanation Strategies

Given a question, the self-explanatory user interface needs to retrieve the necessary information from the underlying models of the user interface, and compose the answer based on that information. This is the role of the *Explanation Strategies* presented in this section.

An explanation strategy is responsible for computing the answer that corresponds to one specific type of question. In consequence, different types of questions are then managed by different explanation strategies.

All the explanation strategies are illustrated through a real example. This example is a car shopping website (figrue 4.12). We will explore the models of the user interface of such a website for each of the different explanation strategies.

4.4.1 Determining the Appropriate Explanation Strategy

As previously described in the Global Approach, we state that each type of question can be answered by one specific explanation strategy. We define then different explanation strate-

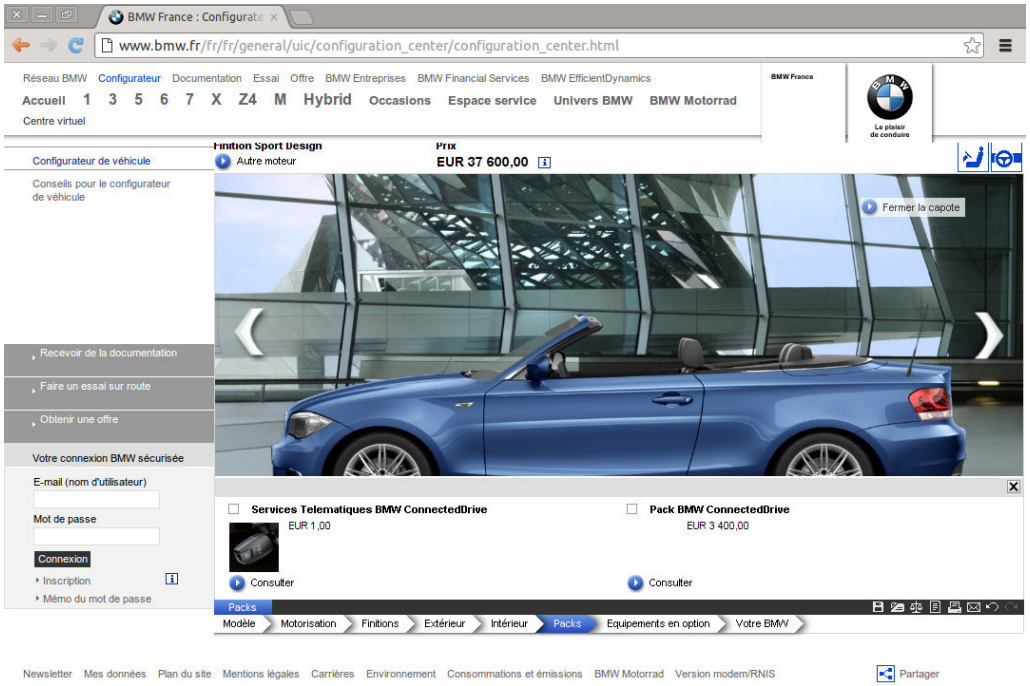


Figure 4.12: The car shopping website example.

gies, one for each of the different types of questions that we want to be supported by our self-explanatory help system. These questions are also generated by the self-explanatory facility.

We describe in the following different explanation strategies for different types of questions. We currently support six different types of questions that have been reiteratedly used by one or more approaches as shown in the state of the art. We built all these explanation strategies upon the main models of the Cameleon Reference Framework. The question types are:

Procedural answering *How* questions.

Purpose or Functional, that provides feedback about *What is it for* questions.

Localization that replies to *Where* questions.

Availability answering the question *What can I do now.*

Behavioural explaining *Why/Why not* things happen in the user interface.

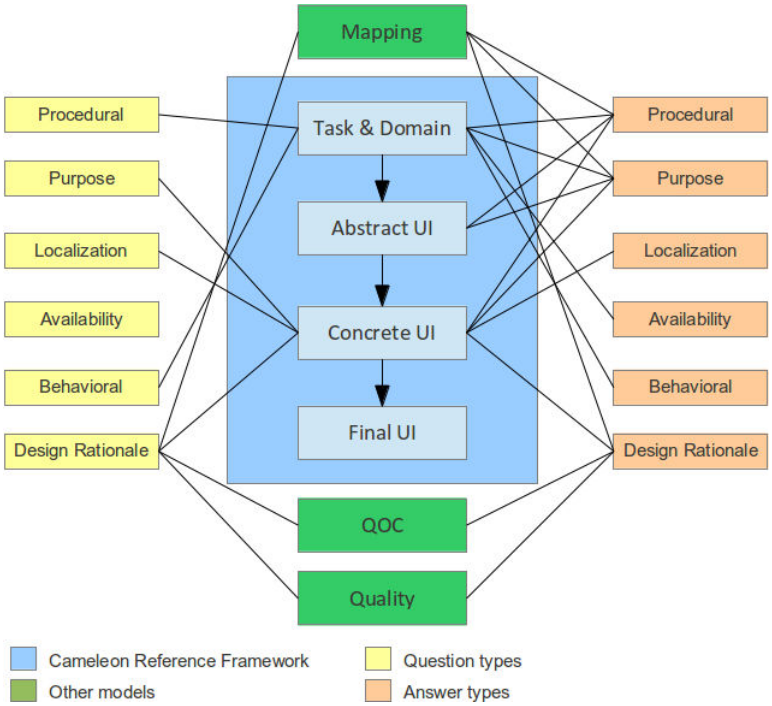


Figure 4.13: Models used for generating questions (left) and answers (right).

Design Rationale that answers questions about the design rationale of the UI.

Once the user asks a specific question, the self-explanatory facility will automatically retrieve the type of the question to determine which explanation strategy needs to be launched, and thus, what models will be inspected.

Figure 4.13 gives an overview of the different models that are involved in the generation of the questions with their respective answers. For instance, procedural questions such as "How to select a car?" are all generated using the task model, which is represented with the link between the Procedural box on the left side of the image, and Task Model in the centre. Answers to procedural questions are computed by using elements of the Mapping, Tasks, Abstract UI, and Concrete UI models, as represented in the image with the four links from the procedural box on the right side of the image to each model in the centre.

In the following, we detail the different explanation strategies that we have developed for each of the six question types. For each of them, we provide an explanation of how the ques-

tions of such type are generated, how the answers are computed, and we provide for each a graphical visualisation of the explanation strategy along with a UML sequence diagram describing the process of generating the answers. The procedure is also documented with an example.

4.4.2 Procedural Questions - How

This section explains how to develop an explanation strategy to support *How* questions. *How* questions are requests that ask for the way in which a task can be accomplished. For instance, for the car shopping website a user can ask "How to select Packs?". The information that the user expects is the description of the procedure to accomplish the task, in the example, the instructions that show the user how to select different packs for a car.

4.4.2.1 Generating Questions

We use the CTT notation in which there are four kinds of tasks: User tasks, Application tasks, Abstract tasks and Interactive tasks. During the interaction with the system, users perform interactive tasks by using the elements of the UI. In other words, an interactive task is always mapped to one or more interactors at the CUI level during the transformation process. It makes then sense to generate questions of the form:

How to + Task.name + ?

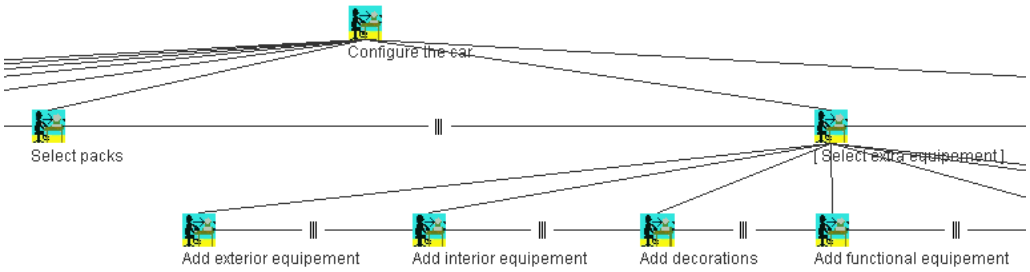


Figure 4.14: Excerpt of the task model of the car shopping website.

where the task is an interactive task. To generate this type of questions, the explanation strategy can then explore the task model recursively from the root to the leaves. For each node representing an interactive task, the explanation strategy creates a question in a textual form according to the previous grammar.

For instance, if we consider the excerpt of the task model of the car shopping website example shown in figure 4.14, the list of questions that are generated by this approach are the following:

- *How to Configure the car?*
- *How to Select packs?*
- *How to Select extra equipment?*
- *How to Add external equipment?*
- *How to Add internal equipment?*
- *How to Add decorations?*
- *How to Add functional equipment?*

These questions are generated because all these tasks are of type interactive, as previously explained.

4.4.2.2 Retrieving Information

According to the task model, an interactive task is always mapped to one or more interactors at the CUI level during the transformation process. Thus, the user needs to interact with such interactors in order to complete the interactive task (from now on, *requested task*). A possible way of answering a procedural questions is then to indicate to the user what are the interactors that he/she needs to interact with in order to accomplish the requested task. A possible way of answering a procedural questions is then by retrieving the interactors in the CUI model, starting from the requested task at the task level.

Figure 4.15 describes the previous process graphically.

The explanation strategy first locates the task inside the task model. Second, it inspects the mapping model that maps tasks to AUI elements from the AUI model, so it can retrieve

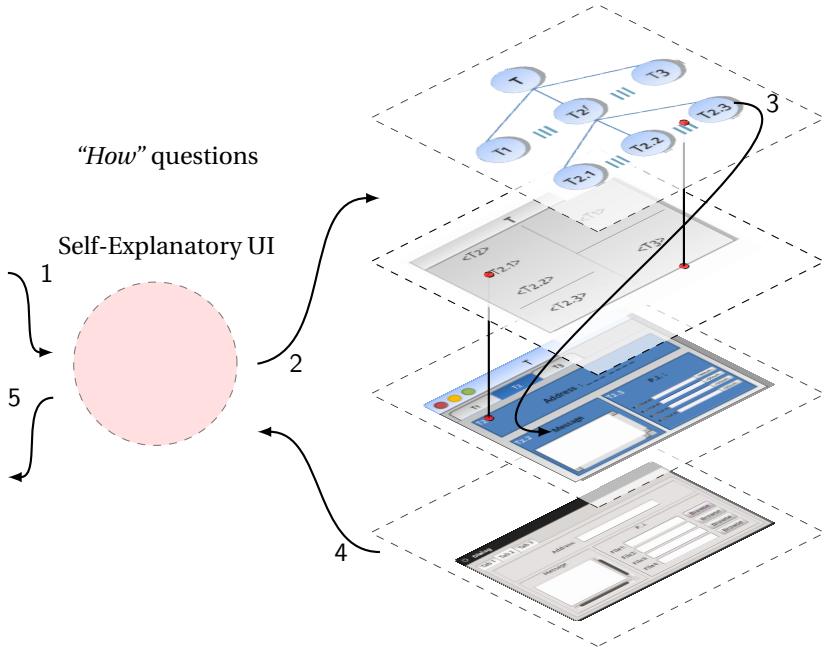


Figure 4.15: Explanation strategy for *"How"* questions. The question identified as *How* (1) is used by the explanation strategy to locate the task of the question (2), then to follow the mappings to reach the widgets at CUI level (3), retrieve these widgets (4) and provide the answer back (5).

the abstract UI elements that resulted from transforming the requested task into abstract user interactors. Once the AUI elements have been found, the explanation strategy repeats the procedure to locate the CUI element derived from the AUI elements. This is done by inspecting the mapping model that keeps track of the transformations of AUI elements into CUI elements. Once the CUI elements have been retrieved, these CUI elements are used to composed the final answer as described in the next section.

As an illustration, consider the models of the car shopping website shown in figure 4.16. At the top of the figure, an excerpt of the task model shows how the "Select packs" task is transformed into its respective AUI element in the middle of the figure. The AUI element called "Select packs" is in turn transformed into a CUI panel which is part of a tabs component. For clarity, the figure only shows those transformations and elements of models that are relevant in this example.

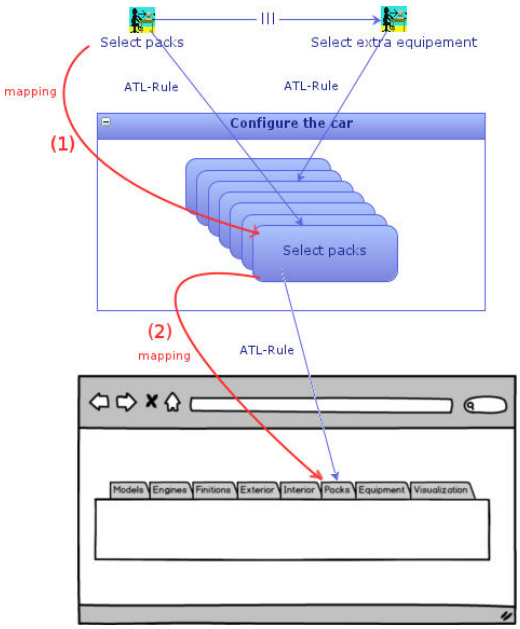


Figure 4.16: Example of information retrieving for *How* questions. Mappings are followed from the Task model to the AUI (1) to find the AUI elements in which the task is transformed. Then, CUI elements are retrieved with same procedure (2).

Every time that a transformation is performed, a mapping between the source and the target is kept according to the mapping model, so the three ATL-rules shown in the figure will generate three mappings that can be in consequence obtained by inspected the mapping model.

The figure 4.17 summarizes the previous reasoning in a sequence diagram.

4.4.2.3 Providing Support

The composition of the answer is done according to the following grammar:

*Use the + CUI-element.name + CUI-element.type [+, CUI-element.name + CUI-element.type]**

By construction, there is always at least one CUI element in which an interactive task is transformed. An example of a computed answer using this approach is:

Use the Packs tab

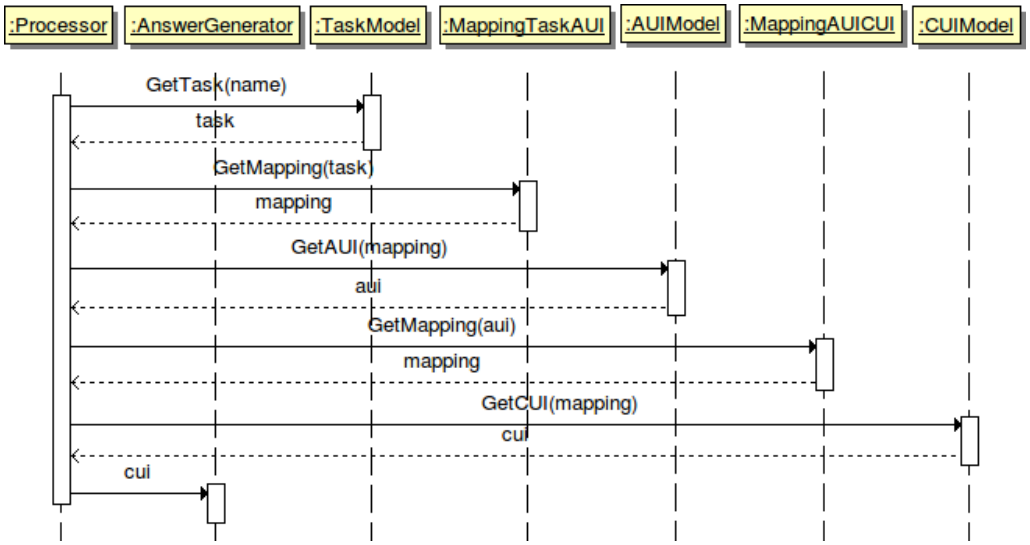


Figure 4.17: Sequence diagram for computing *How* questions

where the *CUI-element.name* is "Packs" and the *CUI-element.type* is "tab".

Note that the answer can be completed with the information about the localization of the widget, which is computed later in the *Where* questions. In this way, a more elaborated answer for CUI elements that were not directly visible from users can be composed as follows:

Use the + *CUI-element.name* + *CUI-element.type* + *in the* + *CUI-element.parent.name* + *CUI-element.parent.type*

where an example is:

Use the Pack Connected Drive checkbox button in the Packs tab

Here, the *CUI-element.parent.name* is "Packs" and the *CUI-element.parent.type* is "tab".

4.4.3 Purpose/Functional Questions - What is it for

In this section we describe an explanation strategy for questions of the type *What is it for*, as well as an algorithm for generating such type of questions at runtime. For instance, the question "What is the Finitions button for?".

This type of question provides information about the goal of a certain component of the UI. The information that the user expects to have is to know what is the utility of such element.

4.4.3.1 Generating Questions

During the interaction with the system, users perform tasks by using the elements of the UI. If we consider graphical UIs we then talk about widgets. The ultimate goal of a widget is to serve for the task in the task model from which the widget has been generated. All the widgets are then suitable to be used for questions of this type. According to this fact, it makes then sense to generate the following type of questions for the different widgets of the UI:

'What is the + CUI-element.name + CUI-element.type + for?'

An example of a purpose question is:

What is the Optional Equipment button for?

To generate these questions we then explore the CUI model recursively from the root to the leaves. For each node representing a widget, we create a question in a textual form according to the previous grammar.

For instance, consider the excerpt of the CUI model of the car shopping website in figure 4.18.

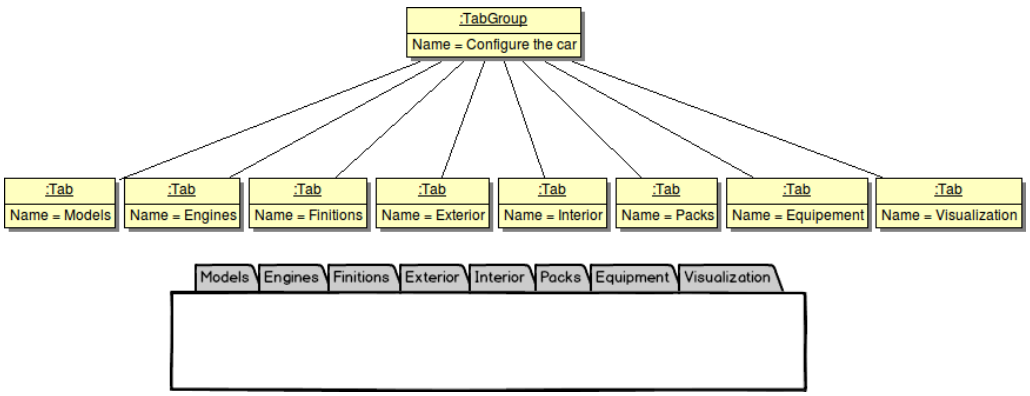


Figure 4.18: Excerpt of the CUI model of the car shopping website described in UML (top) and represented using a mockup (bottom).

The described algorithm for generating questions will generate the following list of questions for the CUI elements of the excerpt:

- *What is the Configure the car tabgroup for?*
- *What is the Models tab for?*
- *What is the Engines tab for?*
- *What is the Finitions tab for?*
- *What is the Exterior tab for?*
- *What is the Interior tab for?*
- *What is the Packs tab for?*
- *What is the Equipment tab for?*
- *What is the Visualization tab for?*

It is worth mentioning that there exists different widgets that users are usually not aware of. For instance, Layouts structuring the widgets inside a Window are not visible for users, so designers can optionally skip these widgets in the generation of questions.

4.4.3.2 Retrieving Information

We can consider this type of question as the opposite of the previous type *How*. In *How* questions, users are asking about elements at a higher level of abstraction (tasks in the task model), whereas in this case we are being asked about the purpose of elements of a low level of abstraction (CUI elements from the CUI model). In the first case, we needed to retrieve CUI elements from a given task. Now, we need to follow the inverse path to discover the task from which a CUI element has been generated, which is the reason of the widget existence.

Figure 4.19 describe this process graphically.

As an example, consider the models of the car shopping website illustrated on figure 4.20. From the bottom at the CUI level, the explanation strategy firstly retrieve the CUI element from the CUI model. It then inspects the mapping model between the AUI and the CUI models, to retrieve the AUI element from which the AUI element has been generated. Once the AUI

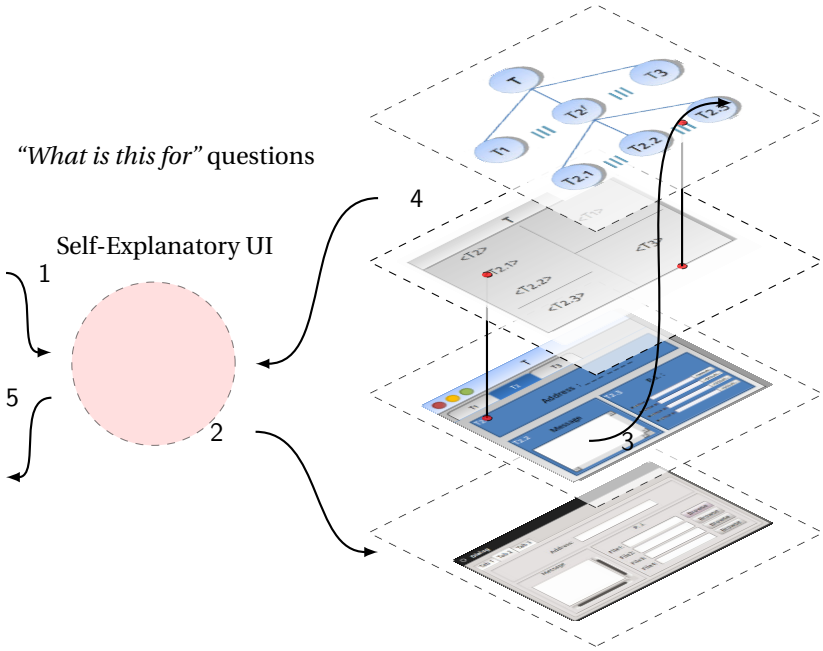


Figure 4.19: Explanation strategy for *"What is this for"* questions. The question identified as of type *Purpose* (1) is used by the explanation strategy to locate the CUI element asked in the question (2). Once the CUI element has been located, the explanation strategy follows the mappings to reach the task at the Task level (3), retrieving the task (4) and providing the answer back to the user (5).

element has been retrieved, the explanation strategy searches for the task originating this AUI element, i.e., the source of the transformation chain, once again by travelling the mappings from AUI to Tasks.

The previous reasoning is summarized in figure 4.21, which details the sequence diagram for computing answers for this type of questions. We can clearly see how these questions are computed in an opposite way to the *How* questions.

4.4.3.3 Providing Support

The composition of the answers for these questions is as follows. Once the interactive task has been retrieved, the explanation strategy directly provide the name of the task as an answer. For this purpose, the following grammar is proposed:

To + task.name

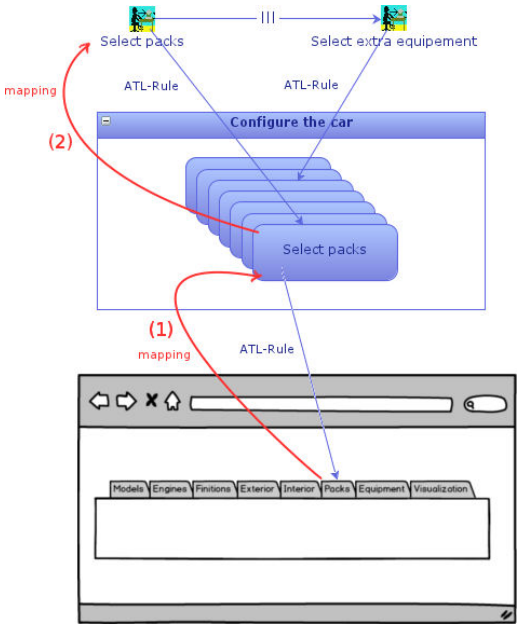


Figure 4.20: Information retrieving for *What is it for* questions. Mappings are followed from CUI to AUI (1) to retrieve the AUI element. Then, from AUI to Tasks (2) to find the task at the source of the transformation chain.

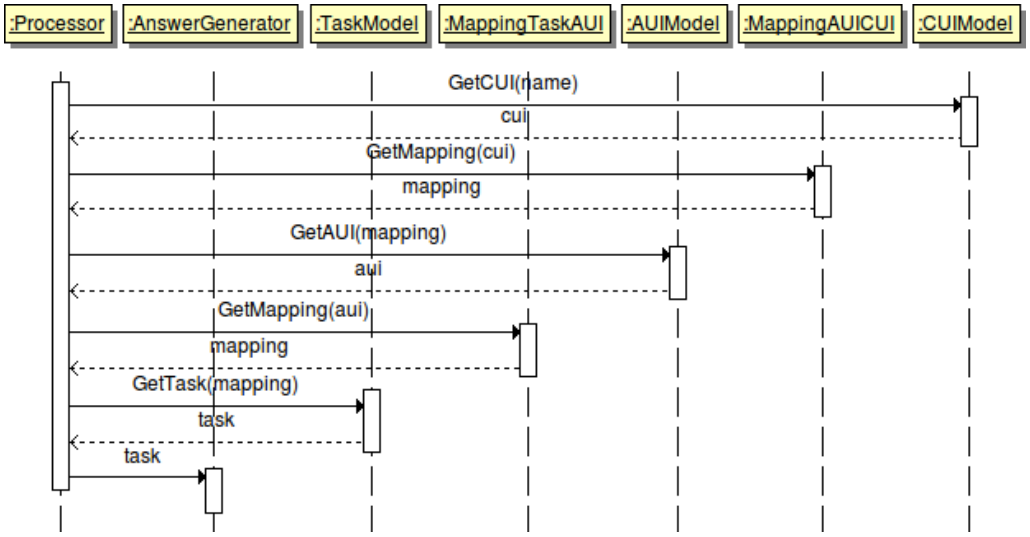


Figure 4.21: Sequence diagram for computing *What is it for* questions

For instance and following with the example of the Packs tab shown in the previous image, the generated answer is:

To Select Packs

Even if this question is mostly useful for images or icons that have an unclear meaning, due to the uniformity of the approach, the explanation strategy can also generate questions and answers for the rest of the CUI elements, even if they are presented with textual information that made clear the purpose of the object (such a label) as in the example covered here.

4.4.4 Localization Questions - Where

We now detail how to generate questions of type *Where* as well as an explanation strategy able to answer this type of questions. *Where* questions aim at localize into the UI a desired element that the user is looking for. This question makes the assumption that the user knows what is the element he/she wants to locate, so the user can ask about where this element is. For instance, if we talk about graphical UIs, users can ask about the location of icons, options, or any other kind of widget. Following with the car shopping website example, a user can for instance ask the question "Where is the Non-Smoker kit?".

The information that the user expects to get is the exact localization of the desired element in the UI. For the previous question, the user expects to get the exact location of the Non-Smoker option in the user interface.

4.4.4.1 Generating Questions

During the interaction with the system, users perform tasks by using the elements of the UI. It is reasonable to consider that users can ask *Where* questions about CUI elements that they see, they have seen, or they know that exist in the UI. Because of this, it makes sense to generate questions for all the elements of the CUI model for which the user can interact with.

According to this, we can then propose the following grammar for the generation of *Where* questions:

Where is the + *CUI-element.name* + *CUI-element.type* + *?*

As in the example:

Where is the Non-Smoker kit?

As an illustration, consider the excerpt of the CUI model of the car shopping website presented in figure 4.22.

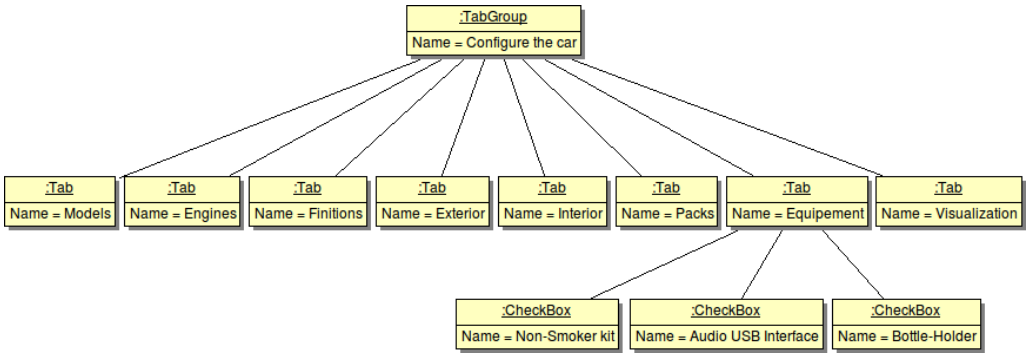


Figure 4.22: Excerpt of the CUI model of the car shopping website.

According to this excerpt, the list of *Where* questions that the previous algorithm generates is the following (from bottom to top):

- *Where is the Non-Smoker kit checkbox?*
- *Where is the Audio USB Interface checkbox?*
- *Where is the Bottle-Holder checkbox?*
- *Where is the Models tab?*
- *Where is the Engines tab?*
- *Where is the Exterior tab?*
- *Where is the Interior tab?*
- *Where is the Packs tab?*

- *Where is the Equipment tab?*
- *Where is the Visualization tab?*
- *Where is the Configure the car tabgroup?*

Readers should notice that, as well as in the previous case for *What is it for* questions, Layouts elements in graphical UIs could be skipped when generating *Where* questions, because users are usually not aware of Layouts and in consequence, it makes no sense for them to ask where these elements are in the UI. But again, the approach presented here is completely uniform and it generates these questions about Layouts as well. This could be interesting for instance if we want to provide support for designers instead of end-users, localisation information about Layouts could be considered as relevant.

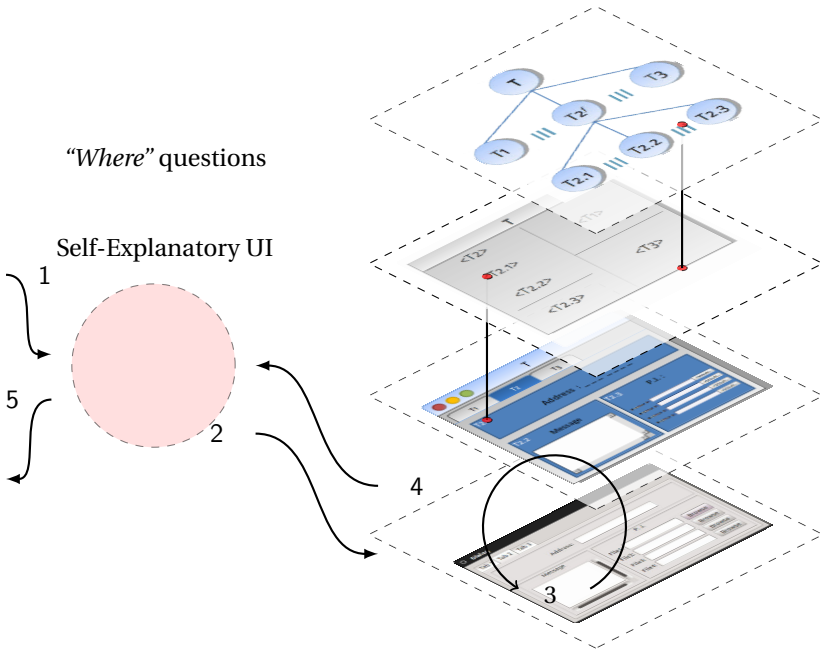


Figure 4.23: Explanation strategy for *"Where"* questions. As the question is identified as of type *Where* by the help facility (1), it is used by the explanation strategy to locate the CUI element involved in the question (2). Once the CUI element has been located, the explanation strategy retrieves the container CUI element inside the same CUI element (3). Once the container is retrieved (4) its information is used by the help facility to compose the final answer (5).

4.4.4.2 Retrieving Information

If the asked question is identified as of type *Where*, the explanation strategy defined for this type of questions acts as follows. Firstly, the explanation strategy locates the CUI element involved in the question inside the CUI model. When the element is located, the explanation strategy explores the CUI model to identify the container element. This could be a panel, a window, a toolbar, or any other container widget in the case of Graphical CUI models. Once the container is retrieved, it is used by the help facility to compose the final answer. Note that according to the CUI meta-model there is always one exact parent for all the CUI elements.

Figure 4.23 describes the process graphically according to our global approach.

Following with the car shopping website and considering the excerpt of the CUI shown in figure 4.24, the parents of a CheckBox inside a Tab element is the Tab element itself. In the same way, this Tab element is contained into a TabGroup container.

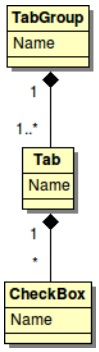


Figure 4.24: Excerpt of the CUI of the car shopping website. All the CUI elements such as Tabs and CheckBoxes have one parent container by construction.

Figure 4.25 details the sequence diagram for computing answers for questions of type *Where*.

4.4.4.3 Providing Support

Once that the CUI element has been located, and its parent container retrieved, the composition of the answer is done by the explanation strategy according to the following grammar:

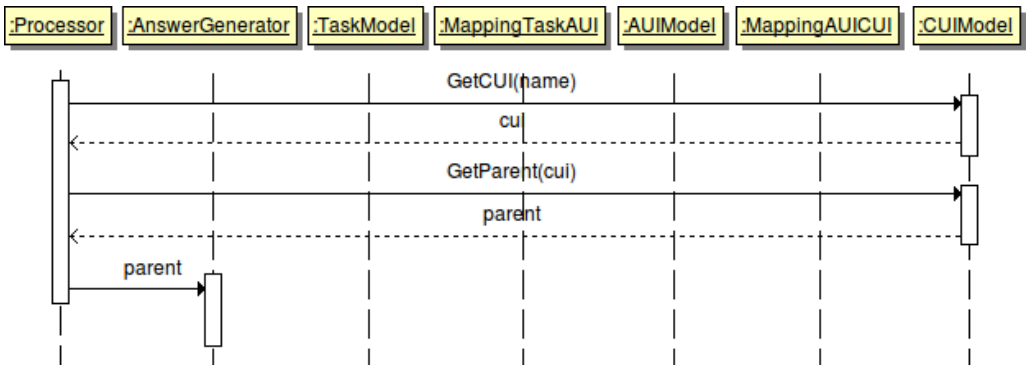


Figure 4.25: Sequence diagram for computing *Where* questions

'The + CUI-element.name + is on the + CUI-element.parent + CUI-element.type'

So for instance, the answer given by the self-explanatory facility to the question "Where is the Non-Smoker Kit?" (according to figures 4.22 and 4.24 is:

The Non-Smoker kit is on the Equipment tab

where *CUI-element.name* is "Non-Smoker kit", *CUI-element.parent* is "Equipment", and the *CUI-element.parent.type* is "tab".

4.4.5 Availability Questions - What Can I Do Now

This section explains how to develop an explanation strategy to support *What can I do now* questions. This type of questions are requests that ask about what the tasks currently available to the user are, regarding the user's current situation in the UI, i.e., depending on the current task that the user is currently performing at the moment of asking the question.

The information that the user expects to obtain by asking this type of question is the list of available tasks that can be performed at the time of asking. For instance, in the car shopping website, when the user is interacting with the equipment tab as shown in figure 4.26

the possible list of tasks for configuring the equipment, including the selection of embedded equipment, the selection of wheel rims, or the selection of maintenance contracts.

4.4.5.1 Generating Questions

The *What can I do now?* question provides information about what tasks are currently available to the user regarding its current situation in the UI. Depending on the task that the user is performing at the time of asking, some tasks will be available and others will not. As not all the tasks are always available in every moment, answers for the same question can vary in time. The presented question is then always of the form:

What can I do now?

4.4.5.2 Retrieving Information

The computation of the answer relies on the task model. The explanation strategy firstly retrieve the current task in the task model. As the task meta-model is based on the CTT notation, we find the available tasks as follows. We firstly locate the current task in the task model. This task model is always tree-form by construction because of the definition of the meta-model. Once the current task has been located into the tasks tree, the explanation strategy computes

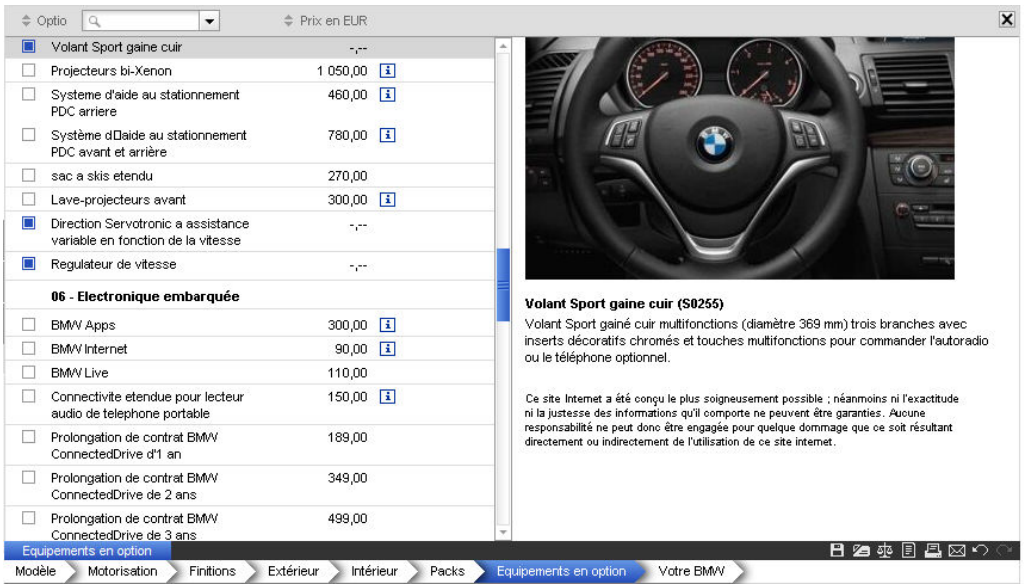


Figure 4.26: Configuration options under the Equipment tab.

what sister tasks are available regarding the LOTOS operators used by CTT. This is done as follows:

- If the task is preceded by the operators of Order Independence, Interleaving, or Choice, the sister is added as an available tasks.
- For each sister identified as available, if it is connected to other sister tasks with the same operators, this tasks are also added recursively as available.

Consider the excerpt of the task model of the car shopping website shown in figure 4.27. In this figure, available tasks are highlighted with a square. Giving a current active task "Select extra equipment", the sister on the left, called "Select packs" is also available because both tasks are connected through the binary operator of Interleaving. By repeating the process recursively, all the right and left sisters are successfully identified as available.

The exploration of available tasks continues by recursively iterating from the current task to the root task of the tree, adding the rest of available tasks, as well as to the leaves of the task tree. As an illustration, consider the previous image, where the current task "Select extra equipment" is subdivided into several tasks (5 shown in the figure). Applying the same algorithm to the children, all the leaves of the task tree are correctly identified as available.

All the tasks that have been identified as available by the previous algorithm, are used in the next step to compute the answer that is provided to the user.

The figure 4.28 describes graphically the general process of requesting this type of information, providing the general perspective according to the Cameleon models. This procedure is also provided in the form of sequence diagram for the convenience of the reader (figure

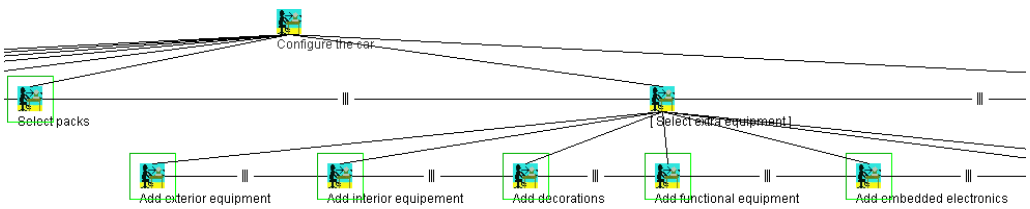


Figure 4.27: Available tasks at the state shown in figure 4.26.

4.29).

4.4.5.3 Providing Support

The *Answer Generator* receives the list of tasks that are currently available in the system. The final answer is then composed simply by listing all the elements of the list according to the next grammar:

You can + task-1.name + . . . + task-N.name

For example and following with the car shopping website, when the user access to the Equipment tab, the answer to the question *What can I do now?* according to the excerpt of

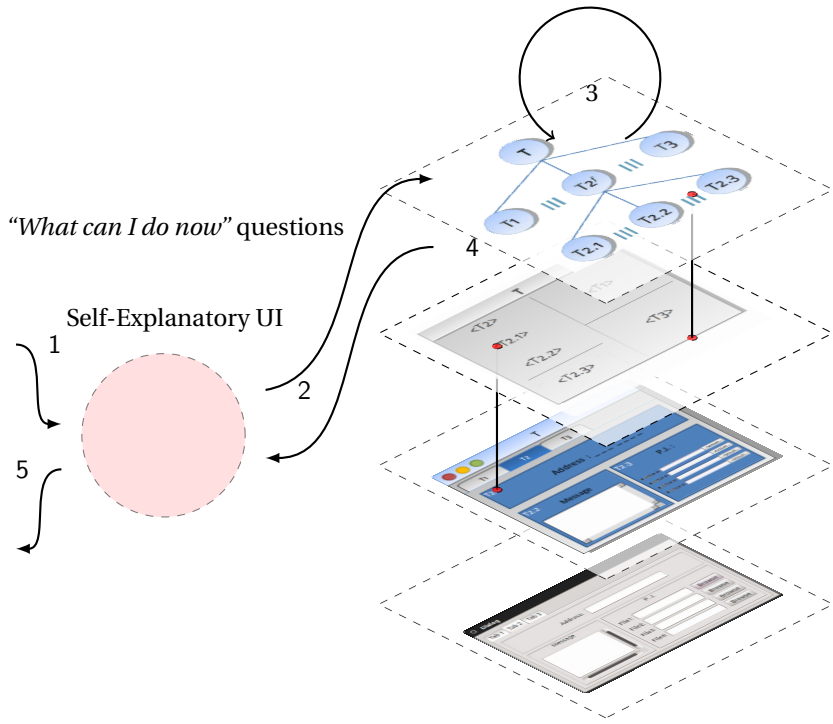


Figure 4.28: Explanation strategy for *"What can I do now"* questions. The question is identified as of type *Availability* by the help facility in (1). The explanation strategy locates the current active task inside the task model (2). The algorithm for finding all the currently available tasks is then applied (3). The list of available tasks is recovered by the explanation strategy in (4). Finally, the help facility composes the final answer (5).

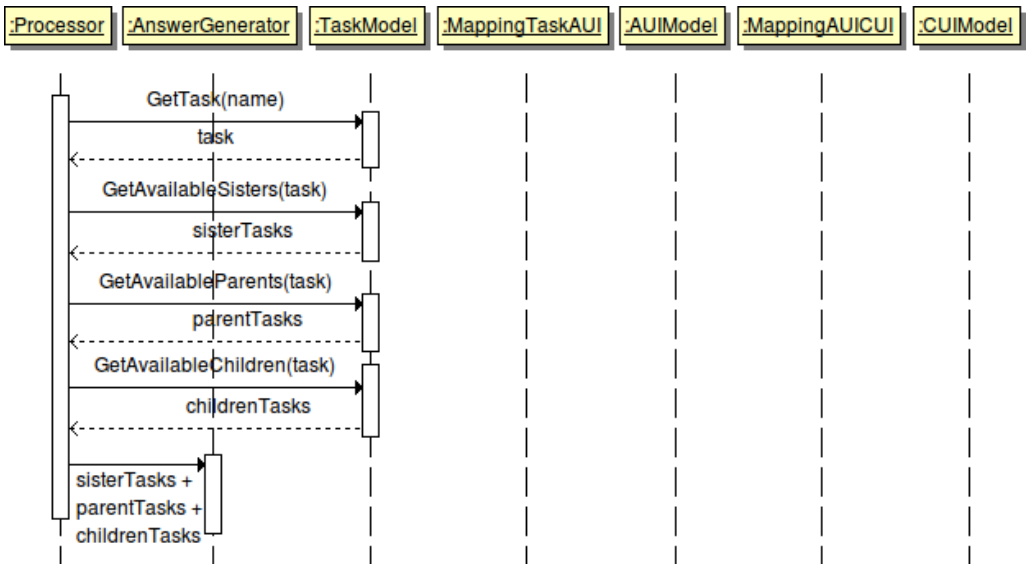


Figure 4.29: Sequence diagram for computing *What can I do now* questions

the task model shown in figure 4.27 is:

> *You can Select packs, Select exterior equipment, Select interior equipment, Add decorations, Add functional equipment, Add embedded electronics.*

Where *Select packs, Select exterior equipment, Select interior equipment, Add decorations, Add functional equipment*, and *Add embedded electronics*, are all the available tasks computed by explanation strategy in the previous step.

4.4.6 Behavioural questions - Why I can't

This section explains how to develop an explanation strategy for *Why I can't* questions. *Why I can't* questions are requests about why the user cannot achieve a specific task in the user interface. It is normally related to disabled options or unexpected behaviour of the application, this is, the user expects something to happen in the UI but it does not occur. The information that the user expects to get is the reason of why he/she cannot do the specified task.

Consider for instance the situation described in figure 4.30. The figure shows the same panel of the car shopping website in different states. The left side of the image presents the

result that the user encounters when the selected car is not shown as expected. The right side of the image shows the selected car when the user has correctly chosen the needed options.

A user facing the unexpected behaviour shown in the left side of the figure, could ask to the user interface “Why I can’t Visualize the car?”.

4.4.6.1 Generating Questions

A mean for generating *Why I can’t* questions is to inspect what are the tasks inside the task model that are somehow unreachable, i.e., those tasks that require a certain input to be activated. For instance, considering the example of a task A that needs some information from the task B. This is represented in CTT with the following notation:

A []» B

This expression involves the operator *Sequential Enabling with Information Passing* as defined in the CTT notation. This means that the task B cannot be started until the task A has finished and it has provided the required data to task B. Due to this, B will be disable or unreachable in the UI until that data arrives. For this reason, a possible question is *Why I can’t do B?*

Following with this line of reasoning, a general approach for generating this type of questions is to look for those LOTOS operators into the task model that produces the previous

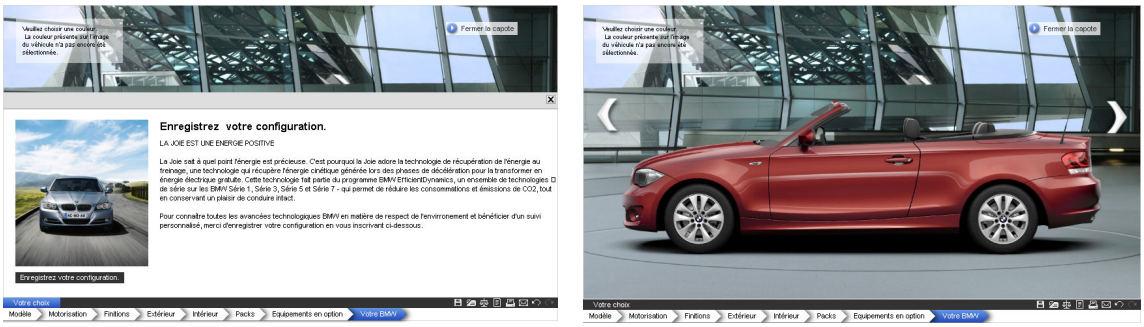


Figure 4.30: Unexpected behaviour (left) and expected result (right) of the Visualization tab. The user gets no feedback about why the unexpected behaviour happens.

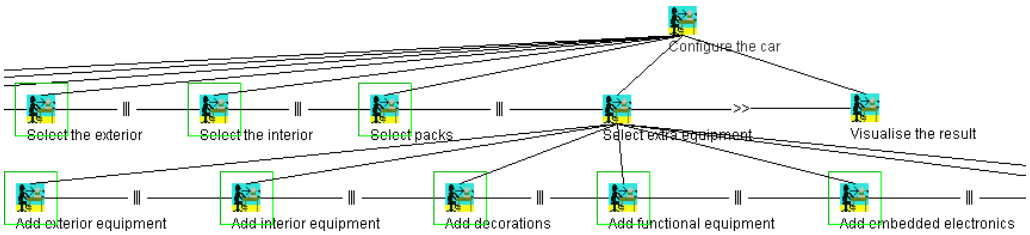


Figure 4.31: Excerpt of the Task model of the car shopping website. The task "Visualize the result" is enabled by all the previous sisters tasks because of the *Sequential Enabling* operator represented by "»".

situation. The explanation strategy that proposed here consider the *Sequential Enabling* operator represented by the symbol "»", and the *Sequential Enabling with Information Passing* represented by "[]». Every time that the explanation strategy finds one of these operators, a question is composed with the task on the right side of the relationship.

The composition of these questions follows the next grammar:

Why I can't + task-N.name + ?

Where the task task-N is an unreachable task at some instant as previously defined.

Considering again the task model of the car shopping website, where an excerpt is shown in figure 4.31, we observe that the task called "Visualise the result" is enabled by all the previous sister tasks. These tasks are highlighted with squares. In this example, the algorithm will find that this task is unreachable because the operator, so the next question can be produced:

Why I can't Visualize the car?

This question is also the unique question generated by the model presented in 4.31, as the rest of operators are not of the specified types.

4.4.6.2 Retrieving Information

We consider the following explanation strategy to compute answers to *Why I can't* questions. For these questions, only the task model is exploited exactly as we did in the previous case for the question *What can I do now?*. The condition for a task to be unreachable is to have either

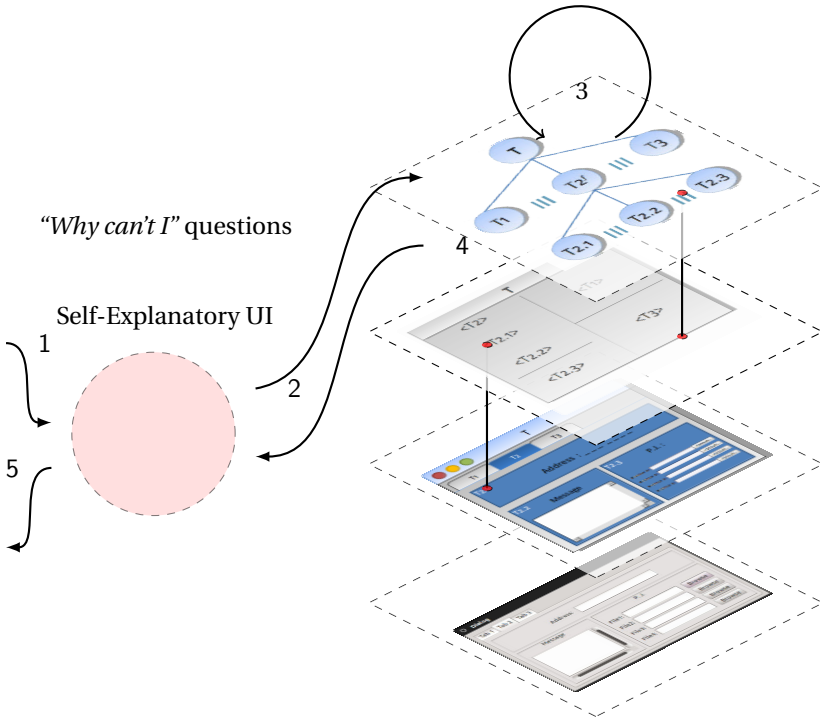


Figure 4.32: Explanation strategy for *"Why I can't"* questions. Questions of type *Behavioural* are treated (1) by the explanation strategy by locating the disabled task inside the task model (2). The algorithm for finding the LOTOS operator that activates this task is then applied (3). This information is recovered by the explanation strategy (4) and the help facility uses it to compose the final answer (5).

a *Sequential Enabling* operator or a *Sequential Enabling with Information Passing* operator on the left side. In addition, if we look at the task model we notice that all the elements of a model, being either a task or a binary operator (operator between two sister tasks), have always one unique parent. With this information, we realise that we only need to travel the task model recursively to look for binary operators, and check whether the operator is of the type *Sequential Enabling* or *Sequential Enabling with Information Passing* or not.

However, a task can be activated as well because a mother task becomes active. The explanation strategy finds these tasks by travelling the mother tasks (up to the root), and locating the LOTOS operators that enables the desired task, i.e., *Sequential Enabling* or *Sequential Enabling with Information Passing* operators.

The final algorithm involves then to travel the sister and mother tasks of the unreachable task until we find the binary operator that enables such task.

After locating the operator enabling the desired task, the explanation strategy locates the task or tasks on the left side of such operator. For instance, in the figure 4.31, all the left sisters need to be done to activate the task "Visualise the result", as a consequence of the operator precedence. The computation of such tasks is done as follows. From the binary operator (the » in the example of the figure), the explanation strategy explores the left task. This task is identified as required, and the explanation strategy inspects the left operator of this required task, if any. If this operator is one of *Order Independence*, *Interleaving* or *Choice*, the left task is marked also as required and the procedure continues recursively. In the case of the example, the "Select extra equipment" task has an interleaving operator on the left side (|||), so the task on the left called "Select packs" is identified as required as well, and so on.

This procedure skips those tasks that are defined as optional.

Figure 4.32 describes the global process of answering *Why I can't* questions in a graphical way, whereas figure 4.33 details the algorithm in a UML sequence diagram.

Once all the operators have been retrieved, the explanation strategy has all the elements for providing the support to the user, which is described in the next section.

4.4.6.3 Providing Support

If a task is not reachable it means that some task or tasks need to be done. The explanation strategy uses the list of tasks enabling the unreachable task with a grammar that answers these questions according to the following construction:

You need to + task-1.name [+, task-N.name]

where at least one task enables the unreachable task by construction, i.e., according to the meta-model[2], all the *Sequential Enabling* and *Sequential Enabling with Information Passing* operators have always a source task (the task that enables) and a target task (the task being

[2]For a detailed description of the meta-models, see appendix B

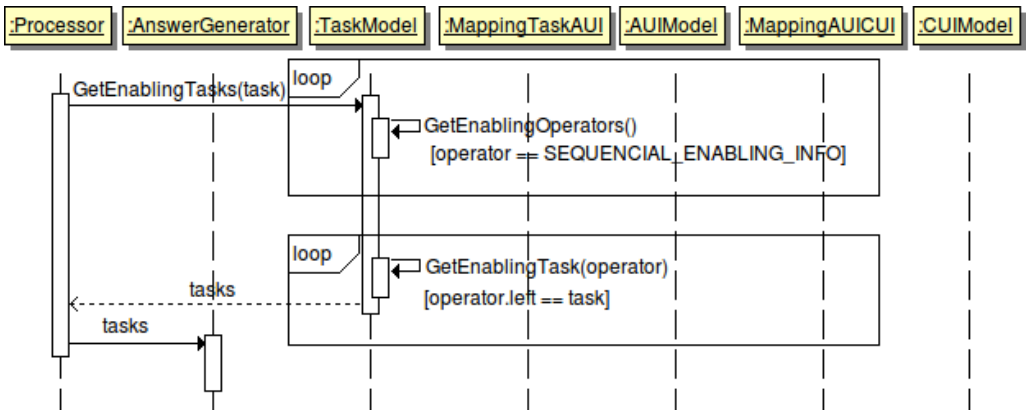


Figure 4.33: Sequence diagram for computing *Why I can't* questions. The algorithm looks for sister and mother tasks enabling an unreachable task.

enabled). For instance, in the case of the task 'Visualize the car', the task was reachable by performing the sister tasks on the left of the operator. The explanation strategy retrieves such tasks with the same algorithm used in the model of the vehicle first. Then, the provided answer by the system is:

You need to Select the model

where "Select the model" is *task-1.name* in the previous grammar.

4.5 Synthesis

The chapter describes a set of conceptual contributions issued from our research.

Firstly, the chapter describes an extension to the Norman's Theory of action through the concept of *Gulf of Quality*. The *Gulf of Quality* couples the perception of designers and users under the same framework.

The chapter then describes the design principles for building model-driven help systems. They are described through a four steps methodology.

The chapter then demonstrates how to define explanation strategies for each type of request. An explanation strategy defines how the necessary information to support users is

retrieved from the knowledge base of the system, i.e., the underlying models of the user interface, and how this information is provided back to the user.

The chapter illustrates these concepts through different explanation strategies that address different types of request for supporting users at runtime. Each explanation strategy can be customized according to the requirements of the user interface under study.

The chapter also describes how to generate a list of possible questions that the system knows to answer. The way in which the list of questions is computed is dependent of the question type, as well as each of the presented explanation strategies.

In total, six different types of questions and answers have been covered through six different explanation strategies. These types are **Procedural** questions answering *How* questions, **Purpose** or functional questions, that provides feedback about *What is it for* questions, **Localization** questions that provides answer to *Where* questions, **Availability** answering the question *What can I do now*, and finally, **Behavioural** questions that explain *Why I can't* questions.

This set of questions belongs to the *Usage* axis of the problem space. The next chapter describes how to address questions that belong to the *Design Rationale* axis.

5

Design Rationale Questions

> *The purpose of models is not to fit the data but to sharpen the questions.*

Samuel Karlin,

RELATED PUBLICATIONS

1. García Frey, A., Céret, E., Dupuy-Chessa, S., and Calvary, G. QUIMERA: a quality metamodel to improve design rationale. In *Proceedings of the third ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2011)* (2011), ACM Press, pp. 265–270
2. García Frey, A., Céret, E., Dupuy-Chessa, S., and Calvary, G. QUIMERA - toward an unifying quality metamodel. In *Congrès INFORSID'11 (Lille, France, May 2011), 6 pages.* (2011)

Poor quality may lead to interaction problems. Bad designed UIs are one of the reasons that increase the Gulf of Quality. Considering quality explicitly in the development of the UIs can drastically reduce the gulf. Those UIs that have been developed with quality in mind, have been proven to be better. For instance, as stated in [2], "results showed that the usability problems identified at this level (FUI level) provide valuable feedback on the improvement of platform-independent models (PIM) and platform-specific models (PSM) supporting the

notion of *usability produced by construction*."

Quality can however be used not only for reducing the Gulf of Quality at design time, trying to assure that the UI that will be produced at the end of the development process will be better because of the quality standards used in such process. Quality can be also used at runtime for answering questions about the UI. For instance, questions about *why the UI is the way it is?* can be supported by the quality criteria behind the decisions made at design time. Moreover, quality can be also ideally used for addressing problems at runtime in those UIs that have been designed without considering quality in the development process. With the aim of using these quality models for explanations purposes, this chapter proposes a quality meta-model. This quality meta-model is not devoted to HCI but, contrary to this, it unifies different quality models of the literature. Moreover, this chapter also explains how quality models can be combined with design rationale notations to keep track of design choices at design time, bringing an argumentation tool that helps designers to create better products in general, and better UIs in the case of HCI. This tandem between quality models and design rationale will be exploited later for explanation purposes.

The chapter begins with the description of the concept of design rationale. It then briefly describes different notations that have been proposed for keeping track of design decisions made at design time.

Secondly, the chapter describes QUIMERA, a quality meta-model to improve design rationale. The chapter explains the different elements of the meta-model, providing also some examples of how to instantiate quality models.

Thirdly, the chapter provides an approach for evaluating different design rationale alternatives through the perspective of the quality. This approach is funded on a design rationale notation and a quality model based on QUIMERA. The approach is illustrated as well with an example.

Finally, the chapter describes an explanation strategy to exploit all these elements to provide user with answers to design rationale questions.

5.1 Design Rationale

Design Rationale is defined in [3] as:

> An explanation of why a designed artifact (or some feature of an artifact) is the way it is.

Different design rationale notations have been proposed for representing design decisions. These notations follow two different approaches.

The first one is an argument-based representation. The first argument-based model is the Issue-based Information System (IBIS) [70]. IBIS uses issues, positions, argument elements and predefined specific relations among them to represent the design rationale. Several subsequent notations were derived directly from IBIS. The Procedural Hierarchy of Issues (PHI) [89], the DR language (DRL) [72], or the POTTS model [120] are some examples of this.

A different approach to capture the design rationale is based on functional representations. A functional representation centres on describing how the device works (or intended to work) [98]. The *Structure, Behavior and Function* model (SBF) [42] belongs to this category.

An example of a QOC model is shown in figure 5.1. The example describes two alternative widgets or interactors for the same task. In this case, designers propose several interactors to let the user enter a date.

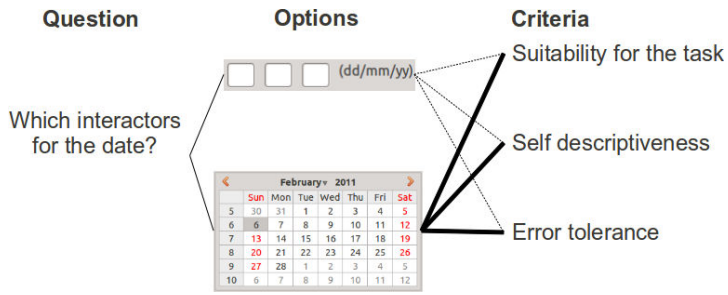


Figure 5.1: Example of a QOC model for choosing between two types of interactors

The first interactor is composed of three input fields, one for the day, one for the month, and a third one for year. A label indicates format notations that the user must respect.

The second interactor is a calendar widget that allows to select a date by clicking instead of typing.

As shown in the figure, the calendar interactor does satisfy the three criteria (assessments links represented with a normal line) whilst the interactor composed of three input fields does not (assessments represented by a dotted line). In particular, the three criteria that we have used in this example are:

Suitability for the task : A dialogue is suitable for the task if the dialogue helps the user to complete her/his task in an effective and efficient manner.

Self descriptiveness : A dialogue is self descriptive if every single dialogue step can immediately be understood by the user based on the information displayed by the system.

Error tolerance : A dialogue is fault tolerant if a task can be completed without erroneous inputs with minimal overhead for corrections by the human user.

The QOC model does not specify what type of criteria designers must use. In our research, and following the notion of *usability produced by construction* discussed in [2], we propose quality as the criteria that helps designers for choosing between different options. For instance, the criteria shown in the previous example are quality criteria extracted from the ISO 9241-110.

For the purpose of this research, we reuse the "Questions, Options and Criteria" (QOC) notation [84]. QOC focuses directly on the discussion between the different design alternatives, making explicit what the design *Questions* are, what are the possible design alternatives or *Options*, and the reasons or *Criteria* used to justify the selection of one of those options among the others.

The main objective of QOC is the discussion of alternatives on specific artifact features. For our purposes, we consider the following elements of the QOC notation:

Options that are artifact features under discussion.

Questions that are means of organizing the various *Options*, since every artifact feature responds to a specific design issue that can be framed as a *Question*.

Criteria that are used to determine the choice between *Options*. Equivalently, they can be seen as requirements or goals that have to be accomplished.

Assessments are links between *Options* and *Criteria*. If they satisfy a *Criterion* then the link is represented with a normal line. If not, a dotted line is used instead (an example is given below).

We have selected QOC because it is the more expressive design rationale representation that works with different alternatives at the same time. Moreover, QOC makes explicit not only the design questions but the reasons justifying the selected option by the means of explicit links.

The use of quality criteria in model-driven approaches requires the construction of quality models that describe quality in terms of quality criteria. Quality criteria can vary from design guidelines to quality standards, covering not only HCI but software engineering in general. With the aim of unifying the different aspects of quality, next section describes QUIMERA, our quality meta-model. This quality meta-model will serve not only for applying quality to model-based approaches but also for extracting design rationale questions based on quality criteria, which will be covered in a later section.

5.2 QUIMERA: The Quality Meta-Model

Software Engineering quality models cover more than usability. They deal with other important aspects of general quality in the whole System Development Life Cycle. The previous chapter 3 reviews different quality models that deal with these aspects. However, whilst several quality models exist in Software Engineering, most of them are oriented to evaluate source code or final products and not models or modelling activities. Other models (for instance, [88, 12, 67]) don't deal with evaluation aspects (evaluation methods, results...) or they just miss the different quality perspectives.

The quality meta-model presented in this chapter has been designed to overcome these problems. Figure 5.2 shows the quality meta-model in detail. To overcome such limitations

and unify the existing quality models under a unique meta-model, our quality meta-model respects the following four basic principles.

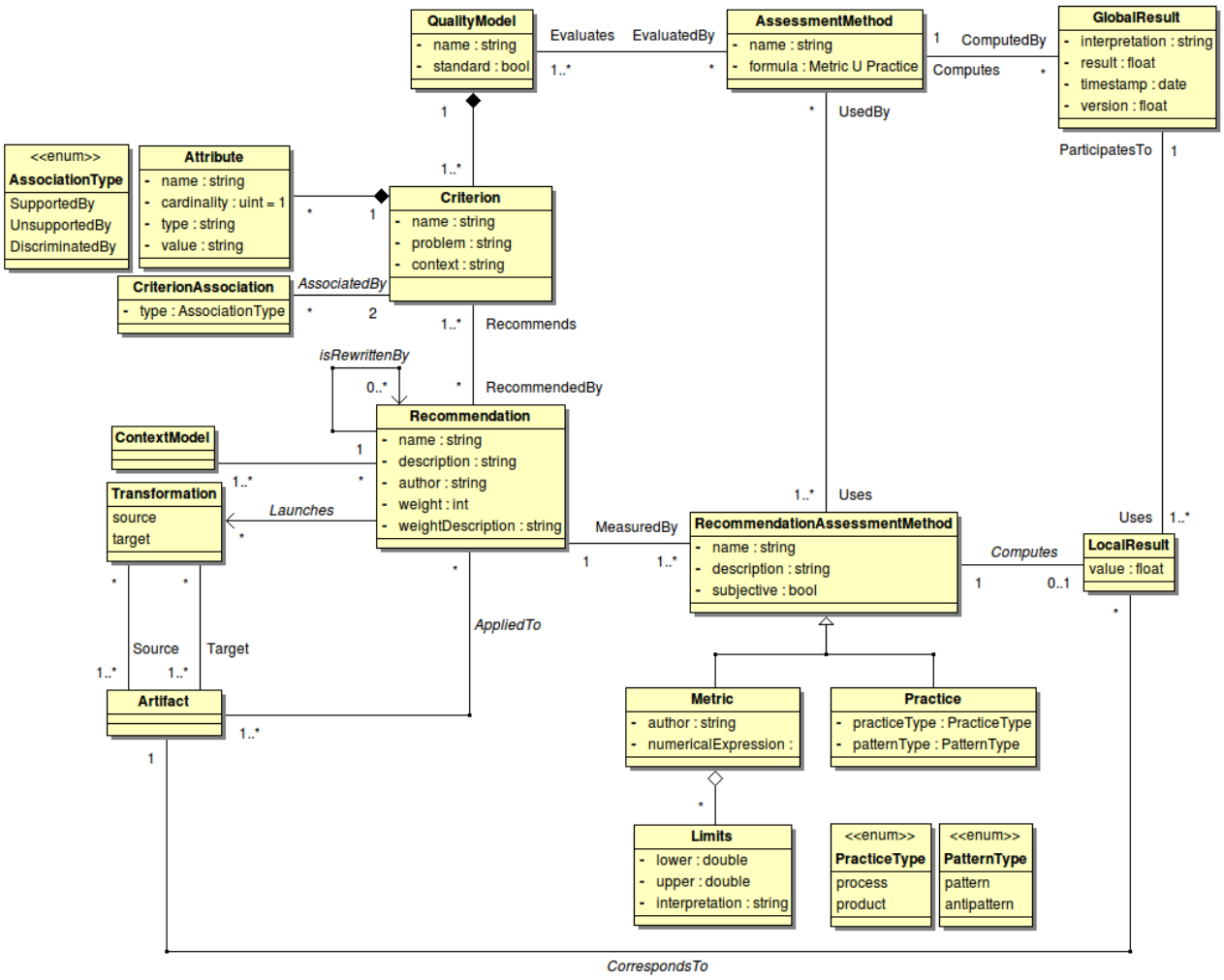


Figure 5.2: Quimera: the quality meta-model

5.2.1 Principles

Our quality meta-model has been designed with respect to the following principles:

1. The quality meta-model must be generic and domain independent.
2. The quality meta-model must be independent of the way in which the measurement is done.
3. The quality meta-model must be independent of the type of quality criteria that composes the meta-model.
4. The quality meta-model must be independent of the way in which argumentation is done.

The first principle means that the quality meta-model is not limited to HCI and, in consequence, instances of the meta-model should permit to represent any quality model that is suitable for any other domains. For instance, one may want to measure the quality of a source code, so a quality model for source metrics can be useful. In this case, the quality meta-model must allow to instantiate this quality model to cover such a need.

The second principle prevents any assumption about the manner in which quality is measured or observed in any instance of the quality meta-model, i.e., in any quality model. In other words, the quality meta-model must permit to its instances to define their own way to produce quality measurements on the system under study. Moreover, the quality meta-model must not force its instances to define quality measurements if they are not desired.

The third principle states that the quality meta-model does not force any specific quality criteria. This has strong consequences in the design of the quality meta-model, because the quality criteria of two different instances of the quality meta-model, this is, two different quality models, can propose a different structure for their quality criteria, and both cases must be covered by the meta-model whatever the structure of the criteria is. For instance, Boehm's quality model [12] decomposes quality criteria into three different levels plus one level for metrics, while the QUIM model [130] distinguishes between four different levels covering factors, attributes, metrics and data.

The fourth principle makes no assumption about how the quality models are linked to the underlying products. To explain this point, consider the previous QOC example in which quality criteria is used to chose between different design alternatives. Designers can choose

one alternative from a group of possible alternatives, regarding which alternative presents a better quality for the user interface. This discussion between different product alternatives based on quality assessments can be captured using different design rationale notations as for instance the QOC model used in this example. The fourth principle states that the previous argumentation about what alternative/s to consider and how quality contributes to each of the alternatives, must not be determined or influenced by the quality meta-model itself, i.e., the quality meta-model does not make any assumption on how this argumentation is done, what design rationale notation is used, if any, and in consequence, how the quality criteria of a quality model is linked to the different alternatives or elements of such alternatives.

A direct consequence of these four principles is that, as the quality meta-model makes no assumption on the purpose of its instances or quality models, these instances should be able to represent any of the four different perspectives of quality. These perspectives are analysed in the following section, as well as how the instances of the quality meta-model relate to them.

5.2.2 Quality Perspectives

Any instance of the quality meta-model, i.e., a quality model, should be able to cover not only the needs of both Software Engineering and HCI, but the four different quality perspectives in which quality can be expressed according to [19]. These four quality perspectives are:

Expected Quality or the quality the client or user needs. It is defined through the specification of the system under study (SUS).

Wished Quality is the degree of quality that the quality expert wants to achieve for the final version of the SUS. It is derived from the Expected Quality.

Achieved Quality is the quality obtained for a given implementation of the SUS. Ideally, it must satisfy the Wished Quality.

Perceived Quality is the perception of the results by the client or user once the SUS has been delivered.

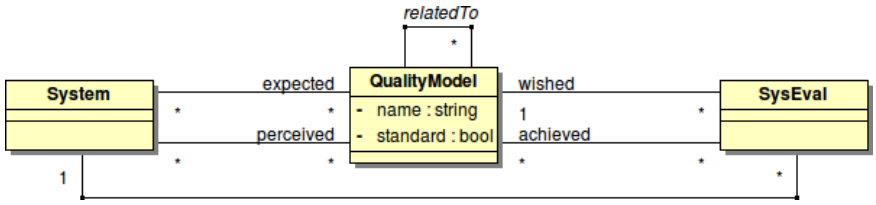


Figure 5.3: Modelisation of Quality Perspectives

Figure 5.3 details how we relate a quality model to each of these perspectives. As stated in [24], these four perspectives can be related to the Systems Development Life Cycle along three dimensions. These dimensions are the *Specification* (related to the Expected and Wished Qualities), the *Implementation* (related to the Achieved Quality) and the *Use* (related to the Perceived Quality). We express these four perspectives with four different relationships (figure 5.3). The *System* entity represents the product for which a quality model is considered as for instance, a user interface. *SysEval* represents a specific evaluation for that product.

We consider these four quality perspectives are four different uses of the same quality model. The attribute *standard* of the *QualityModel* meta-class means that, when true, the quality model is not linked to *System* and *SysEval* as it only represents a quality standard such as ISO9241-110 or QUIM. In other words, the quality of these standards is not defined in terms of a product, and it only represents the desired quality standard. This is useful to express that in a given design process, we have already defined our quality standard but it has not been applied to a specific product yet, so no quality measurements have been performed. This attribute allows also to re-use different quality standards to different products. As a consequence of this, some internal parts of the quality meta-model that are related to the evaluation of a product based on the proposed quality standard (or whatever the quality criteria is), are not necessarily defined when the attribute standard is true. This is detailed in the next section, along with the quality meta-model that allows to define quality models for all these different quality perspectives.

5.2.3 The Quality Meta-Model

Figure 5.2 details the quality meta-model. A quality model is composed of criteria, that can be recursively decomposed into subcriteria through the meta-class *CriterionAssociation*. For instance, in the ergonomic guide from Bastien and Scapin [7] the *Error Management* criterion is subdivided into *Error Protection*, *Quality of Error Messages*, and *Error Correction*.

Different *Recommendations* can be specified for each *Criterion*. A *Recommendation* is a positive assessment that characterizes *Criteria*. For instance, a quality expert can suggest a *Recommendations* for "keeping the complexity of the code low". This recommendation can involve one or more quality criteria.

Different *Weights* can be specified for each *Recommendation* to define which of them are more important than others for the considered system. This allows designers to adjust the global quality precisely.

The meta-model permits to evaluate the quality of a product. To this end, evaluations can be performed through *AssessmentMethods* that are specified by *Metrics* and/or *Practices*. In the first case, the measure is given by a *Result* that can be comprised between some *Limits* when these limits are defined. In the case of *Practices*, the *Result* represents if a practice has been followed with a value of 100% or not (0%). The value of the *Result* can be any intermediary percentage as well. Note that a *Practice* can be either a *pattern* or an *anti-pattern*, applied at the process level, or on a product. *Metrics* and *Practices* are directly evaluated on *Artifacts* through *Recommendations*. These *Artifacts* can be no matter what element of the Software Development Life Cycle, such as code, classes of a model or even the model itself.

Once a quality standard has been defined through criteria, the quality meta-model can be reused with the association *relatedTo*, and extended with several classes such as *AssessmentMethods*, *Transformations* or *Artifacts*, to represent the four quality perspectives. For instance, *Metrics* can be defined in order to obtain some desired values (*Wished Quality*). The importance of every *Recommendation* can be customized using *Weights*. Then, evaluations of the current quality of the SUS can be performed. When a *Result* of an evaluation does not

satisfy the expectations of the quality expert, this is, the *Achieved Quality* does not satisfy the *Wished Quality* (for instance, the *value* for a *Metric* is not within the desired *Limits*), the designer needs to increase the quality. This can be done by setting a *Transformation* or a set of *Transformations*. These *Transformations* are performed on the related *Artifacts* on which the *Result* has been previously calculated. Iterations on this process of adjusting quality by applying transformation on the different elements of the product or *artifacts* can be done until the desired values defined by the quality expert (*Wished Quality*) are reached. *GlobalResult* holds the general quality of a SUS at a given moment. The difference between *GlobalResults* and *LocalResults* is explained in the next section.

5.2.4 Global Quality vs Local Quality

Figure 5.4 shows the different subsets of the quality meta-model regarding *Global* and *Local* quality levels. To explain these levels, three vertical columns make explicit the following information:

Objects that define quality in a concrete and explicit way using specific terms (for instance what criteria is considered for the quality model) and how this quality is structured (for instance, tree-based quality models having factors and metrics).

Methods used to measure the quality of an element in terms of the previous objects.

Results containing the values or output of the previous methods.

Based on these previous elements, we define the terms *Global quality* and *Local quality* as follows:

The **Global Quality** level is the group of Objects, Methods and Results directly focused on the general quality of a SUS.

The **Local Quality** level is the group of Objects, Methods and Results focused on the quality of a given Criterion (and then, all the associated Recommendations).

As shown in figure 5.4, these levels are interrelated to the three vertical columns, making explicit what *Objects* are being measured at the current level, which is the element responsible

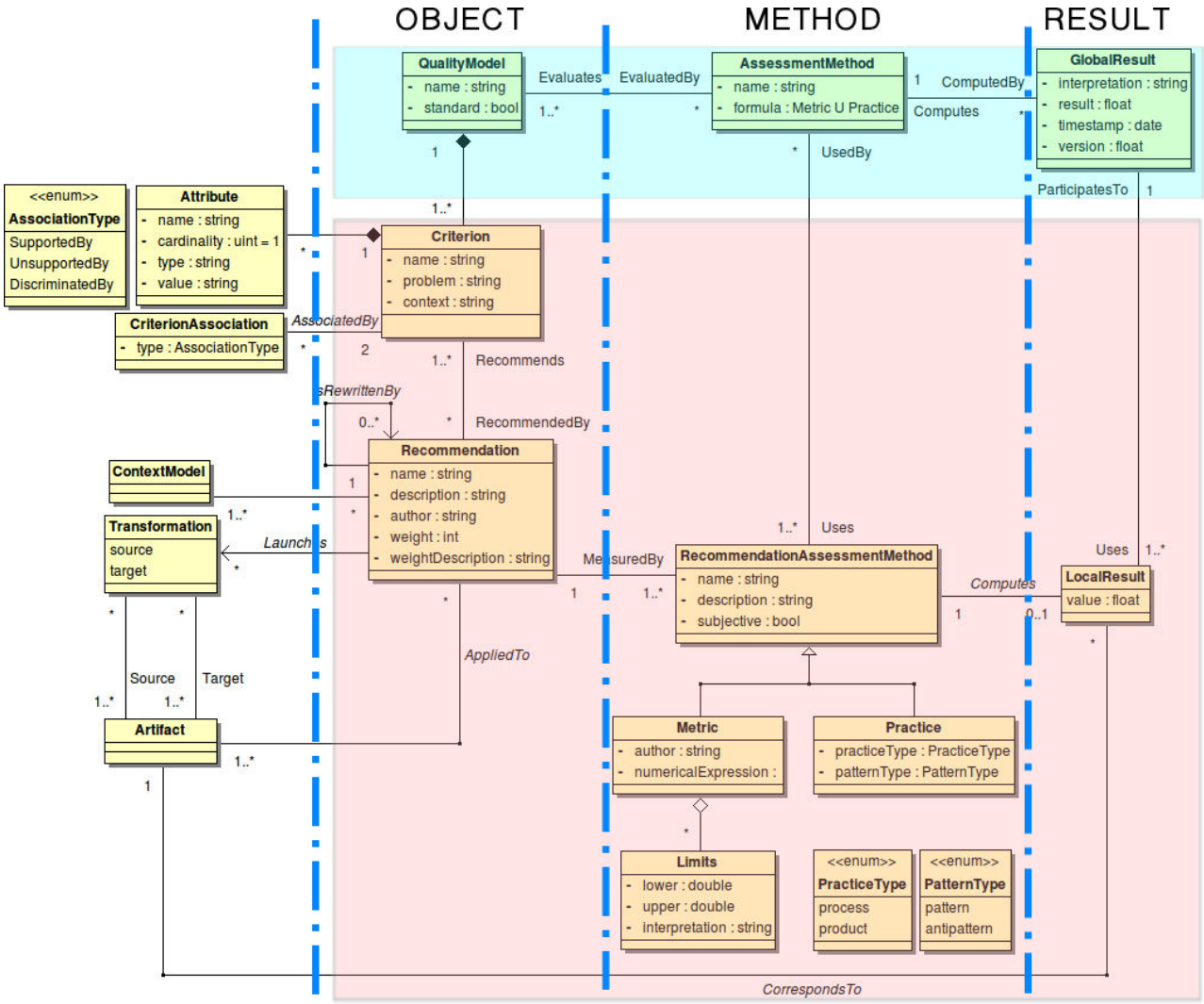


Figure 5.4: Global quality (green top box), Local quality (pink bottom box), and their relationship to Objects, Methods and Results.

of the measurement *Method*, and the quality level of the *Result* for such object.

The Global Quality of a SUS at a given moment according to a *Quality Model* is represented by the *GlobalResult* meta-class, and it is directly computed following the formula described in an *AssessmentMehtod*.

At the *Local Quality* level, the *LocalResult* meta-class represents partial contributions to the quality of the SUS. *Criteria* is evaluated through *Recommendations* by *RecommendationAssessmentMethods* meta-classes, each of them providing one *LocalResult*. All these results are weighed later at the *Global* level. The importance of each *Recommendation* is specified by *weights* that can be used by the quality expert in the *AssessmentMethod* formula.

For a more detailed description of each meta-class and its attributes, please, refer to the appendix A.

The next section describes how to use the quality meta-model to instantiate a quality model, providing examples of instances and describing the instantiation process.

5.2.5 Quality Models: Instantiation Examples

This section describes two different case studies that shows two quality models for different purposes. The first one is applied to HCI. The second one is applied to Software Engineering.

5.2.5.1 A quality model covering the ergonomic criteria in HCI

Figure 5.5 shows an excerpt of a quality model of Ergonomic Criteria in HCI according to the ergonomic rules defined by Bastien and Scapin [7]. The criteria are divided into subcriteria until the final ergonomic rules are derived. As an example, we describe the following three subcriteria:

- Error Protection is a subcriterion of Error Management. It refers to the means available to detect and prevent data entry errors or actions with destructive consequences.
- Minimal Actions is a subcriterion of Workload. It concerns workload with respect to the number of actions necessary to accomplish a task.

- Prompting is a subcriterion of Guidance. It refers to the means available in order to lead the users to make specifications, providing the required formats and values.

A *Recommendation* is a positive assessment that characterizes one or more criteria. Figure 5.5 shows how different *Metrics* are used for the same *Recommendation*. This *Recommendation* says that good quality can be achieved by maximizing the number of criteria that are satisfied by a User Interface. To evaluate criteria, two different *EvaluationMethods* are defined based on different formulas.

On the one hand, the first evaluation method, called *Eval1*, subtracts the number of unsatisfied criteria from the total of satisfied criteria. On the second hand, the second evaluation method, named *Eval2*, counts the number of quality criteria that are satisfied by the system under study.

The next section proposes a different case study where a different quality model is applied to evaluate a design method.

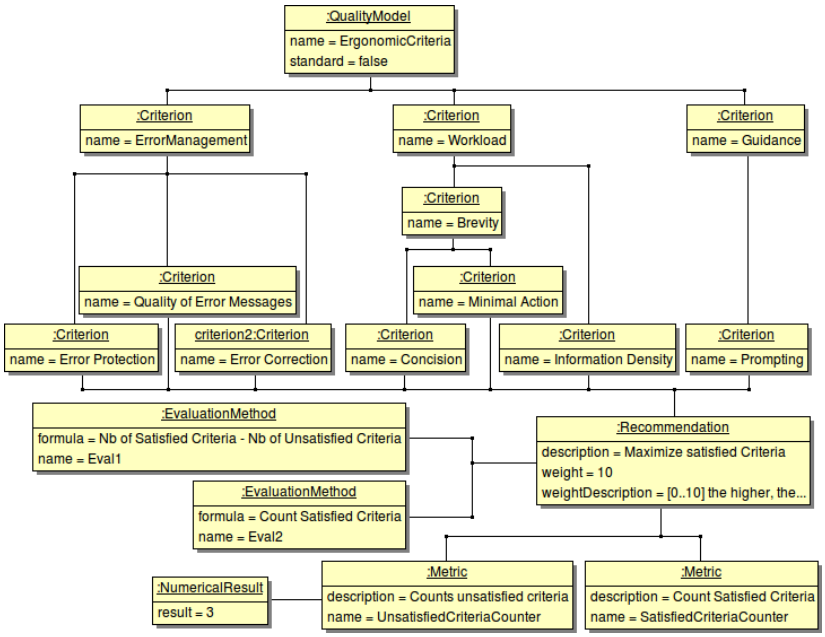


Figure 5.5: A part of the quality model of ergonomic criteria.

5.2.5.2 Application to the evaluation of a design method

Originally developed by the UMANIS Company, Symphony is a method focused on business components. It has been extended to include the design of complex interfaces [51]. Symphony is based on the iterative identification and description of business components. The extension of Symphony supports design of HCI concerns in a similar way: interactional entity objects are basic interactional concepts, i.e. the graphical representation of a concept. Interactional process objects describe the logic of the interactional domain, e.g. the management of an immersive 3D scene.

The purpose of the research described in [22] is to verify that the use of interactional and business objects and the management of communication between all these components improve the final quality of the software. Thus, the quality of several implementations of the same project has been measured and compared, and software quality criteria and metrics have been defined and valued.

We have modelled these criteria and metrics according to QUIMERA. The resulting model contains 39 classes. Figure 5.6 presents a subset of these elements. The figure shows two criteria, reusability and maintainability. These criteria are refined when needed, e.g. maintainability is composed of independence, sizes and complexity criteria. *Recommendations* are associated to criteria: according to [154], we defined that the cyclomatic complexity - the number of linearly independent paths in the code, i.e. the minimum number of paths that should be tested - has to be low so that the code can actually be tested. The different limits for each metric have been modelled: cyclomatic complexity is good when lower than 4, and too high when greater than 11 [22]. The numerical results have been represented and associated to an artifact, here the whole application.

5.2.6 How to build a Quality Model

The previous examples have been built following a similar process. In this section we describe this process, detailing the steps that the quality expert must follow to define a quality model,

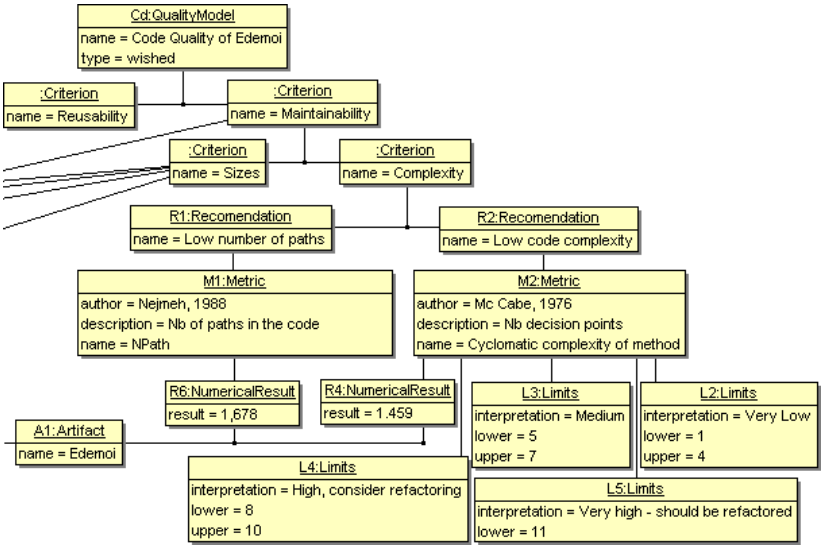


Figure 5.6: Subset of the objects of Symphony evaluation model.

i.e., how to correctly instantiate the quality meta-model. This description involves identifying which classes of the quality meta-model (figure 5.2) are instantiated for each quality perspective. The whole process consist of the following five different phases:

1. Firstly, the quality expert must identify which quality standard is the more appropriate to fit the product requirements, i.e., identify the relevant elements in the specification of the SUS that are related to quality. These requirements are the *Expected Quality*. Once the *Expected Quality* is extracted from the specification of the SUS, the quality expert can now proceed to select the best quality model for such requirements. For instance, an ISO standard, any other quality standard, or even a customized quality model based on the criteria developed by the quality expert.

2. The quality meta-model can be instantiated now to represent the desired quality model for the particular product. For this, the *Criterion* meta-class is instantiated using the *CriterionAssociation* meta-class to structure the *Criteria* conforming to the selected quality model or standard. An example is shown in the right side of figure 5.7. Once all the *Criteria* has been defined, specifying attributes and linking *Criteria* through the *Crite-*

rionAssociation meta-class, the attribute *standard* from the *QualityModel* meta-class is set to true. This indicates that only a standard is represented at this point and no other classes are instantiated yet (such as metrics or transformations). This allows the quality expert to re-use different quality models for other projects.

3. Thirdly, the quality expert can define the necessary recommendations based on different metrics and/or practices, as well as *AssessmentMethods* to allow the system to perform automatic quality evaluations. To do this, the quality expert will turn the *standard* attribute to false and will extend the quality model with all the necessary *Recommendations*, *Metrics*, *Practices* and *AssessmentMethods*. This new extended version of the Quality Model is able to compute the *Achieved Quality* through *AssessmentMethods*. For those *Practices* that cannot be automatically evaluated such as *Antipatterns*, the quality expert can express the necessary *LogicalResults* as, for instance, if an *Antipattern* is present or not.

4. The next step involves the definition of *Limits* of values for the desired metrics in case the system has some. This is done instantiating the *Limits* meta-class for each desired metric. This part of the Quality Model holds the *Wished Quality*, i.e., the values the metrics must ideally reach. This new version of the quality model extended with the definition of all the evaluation related elements, can be also re-used for different products.

5. The last step consists in defining the *Transformation* that will modify the underlying *Artifacts* to increase the quality of the product, when the *Achieved Quality* is not enough. These transformations can apply to one or more different artifacts.

Note that different iterations can be done in order to achieve the expected quality. For instance, if the result of a metric is not achieved, i.e., the value of the *NumericalResult* is not between the limit values, a transformation can be launched (if it has been specified) and performed on one or more *Artifacts* trying to achieve the desired value. Then, the global quality can be recalculated again and compared to the previous quality before the transformation.

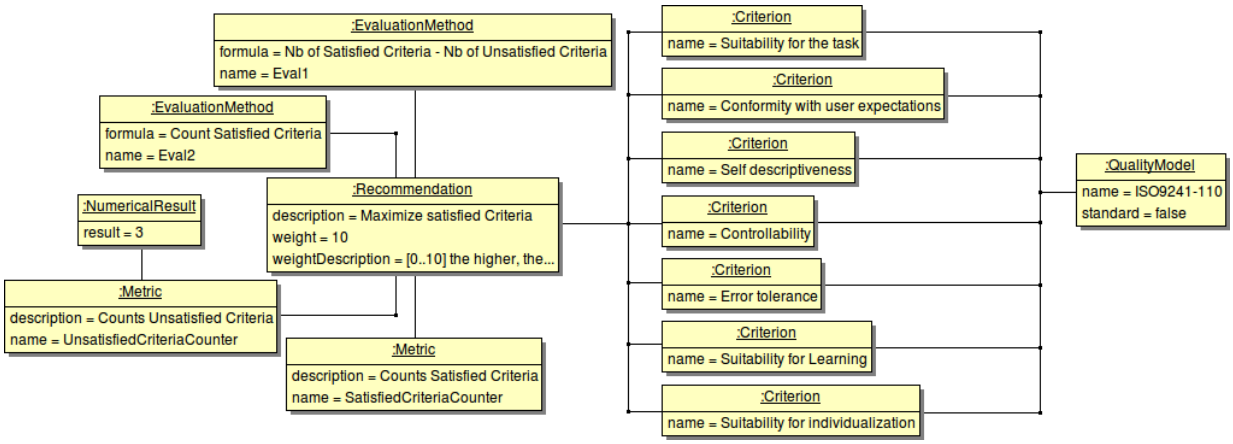


Figure 5.7: Example of an instance of the quality meta-model. The quality model is the standard ISO 9241-110. The instance shows a subset of seven criteria from this standard.

With the quality meta-model QUIMERA and its instances and a design rationale notation such as QOC, we have now all the necessary elements to understand how to link both models and provide explanations for design rationale questions. The next section explains this through an example oriented to user interfaces.

5.3 Design Rationale and Quality

Our proposition is to combine both the QOC model and the quality model by the use of quality criteria from the quality model as the criteria specified in the QOC model. To this end, this section explains the combination of both models and describes how design alternatives can be quantified from a quality perspective. Then, some advantages and limitations of this proposition are presented.

5.3.1 Putting the Pieces Together

The combination of a QOC model and a quality model is shown in figure 5.8.

Note that the quality model is not a merely representation of the ergonomic criteria from the quality standard ISO 9241-110. Ergonomic criteria play different role because it can launch transformations that affect to the system under study. To explain this idea, consider that the quality expert has defined the criteria to be used (a subset of the ISO 9241-110 in this case) as the one shown in previous figure 5.7.

By combining these criteria directly with the QOC model of the example, linking both through assessments, we obtain the result shown in figure 5.8.

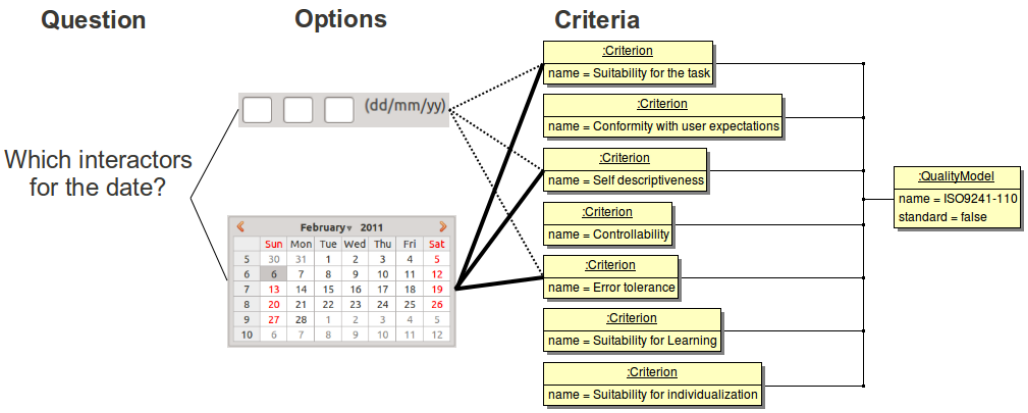


Figure 5.8: Graphical representation of the connection between the quality model (right) and a design rationale notation, QOC in this example (left).

The quality model has all the elements of the comparison between both alternatives, the one with the calendar widget, and the one based on input fields. The comparison can be done according to the *EvaluationMethods* defined by the quality expert that are depicted in the left part of the figure 5.7. These *EvaluationMethods* allow to quantify the quality of both alternatives by using their included formulas for computation of the *LogicalResults*:

$$\text{Eval1} = \text{Number of satisfied criteria} - \text{Number of unsatisfied criteria}$$

$$\text{Eval2} = \text{Number of satisfied criteria}$$

The computation is based on the number of satisfied versus unsatisfied criteria depicted in figure 5.7, so the following computation can be now performed for the first evaluation method:

$$\text{Eval1(Calendar)} = 3 - 0 = 3$$

$$\text{Eval1(Text fields)} = 0 - 3 = -3$$

and for the second evaluation method we obtain:

$$\text{Eval2(Calendar)} = 3$$

$$\text{Eval2(Text fields)} = 0$$

which concludes that the design alternative considering the calendar widget has a better quality than the input fields, accordingly to the three criteria from the ISO 9241-110 that has been selected by the quality expert, and the evaluation formulas *Eval1* and *Eval2* defined for such criteria in *EvaluationMethods* inside the quality model.

At this point, a transformation that chooses the Calendar over the text fields could be launched so that the final product, i.e., the user interface discussed through the design rationale notation, will contain a calendar as the selected widget instead of the other alternative.

This example illustrates how quality in general, and the quality meta-model in particular, can be involved in the design process. Quality experts can take benefit of design rationale questions by directly exploiting quality models through quality criteria, evaluating and adjusting the final product according to their quality expectations.

The next section discusses some advantages and limitations of the quality meta-model.

5.3.2 Advantages and Limitations

The tandem quality-design rationale provides interesting advantages for the designers:

1. Quality in design decisions becomes measurable.
2. Design decisions can be explained directly through quality models.
3. As a design rationale can be directly evaluated, two different solutions can be compared.
4. The quality model provides a mean for adjusting the quality of a system. For instance, as we have shown in the previous example, one transformation could choose the input

fields while other different transformation could select the calendar widget. This affects the local results computed for that specific question, and thus, the global quality of the product is also modified. Thus, our proposition makes explicit ways of achieving the *Wished Quality* of a system, i.e., the global quality of a system can be adjusted by regarding how a transformation increases or decreases the achieved quality.

5. As a consequence of the previous point, adaptation of UIs can be quality driven.

One limitation of the approach is that only one design rationale notation has been proposed. Other different notations that do not have criteria defined in a explicit way, could probably be used as well, but explicit links with the quality meta-model need to be defined first.

Another limitation involves that improving the quality of the different elements of a system separately, does not ensures that global quality of the system will be improved afterwards. In other words, improving the local quality that affects to individual elements does not imply that the global quality of the whole system will increase. This is due to the fact that all the criteria are not necessarily compatible. For instance, the criterion *Error Correction* could imply sometimes to increase the number of *Minimal Actions* needed to accomplish the task. Thus, increasing one, reduces the other, and vice-versa.

With these concerns in mind, the question of how to evaluate a product semi-automatically remains open. The quality meta-model can launch transformations that affect the local and global qualities of the product so, how to search for the best combination of transformations? What is the optimal algorithm? Is this algorithm automatic or semi-automatic?

As for every type of question, design rationale questions are computed by explanation strategies. The next section describes the particular explanation strategy for this type of questions.

5.4 Explanation Strategy

Questions about the design rationale of the UI are questions that ask about the reasons behind the UI itself. We model these reasons as design choices made at design time by the designers of the UI. The information that the user expects to obtain is the reasons of why the UI is the way it is. This section explains how to provide design rationale questions at runtime, as well as how to compute the answer that will be provided back to the user as support.

5.4.1 Generating Questions

Questions are retrieved directly from the QOC model. The explanation strategy reads the QOC model and presents the possible questions to the users of the user interface.

For instance, an example of question about the design rationale of the UI is:

Why the engines are ordered by price?

5.4.2 Retrieving Information

As explained earlier, the QOC model is used by the designers to discuss different design alternatives for the UI. The discussion is sustained by quality criteria that reinforces the arguments of the designers giving and objective point of view that helps to decide which design option between a set of design alternatives is the most appropriate from the perspective of the quality.

As previously described in the section devoted to the QUIMERA quality meta-model, these quality criteria that sustain the design decisions are elements of the quality model (see figure 5.10. They are linked to the QOC model thanks to a mapping model, in the same way that we keep track of the transformations of a task to an AUI element. Thus, to retrieve the criteria that justifies a design choice, the explanation strategy needs to, in first place, retrieve the question in the QOC model. Then, the explanation strategy can follow the mapping that links the question and options with the criteria that supports that design alternative. These criteria

belongs to the quality model. Finally, the explanation strategy can extract the criteria and use them to compose the answer that the user needs.

Figure 5.9 details the sequence diagram for retrieving the criterion that justifies design choices.

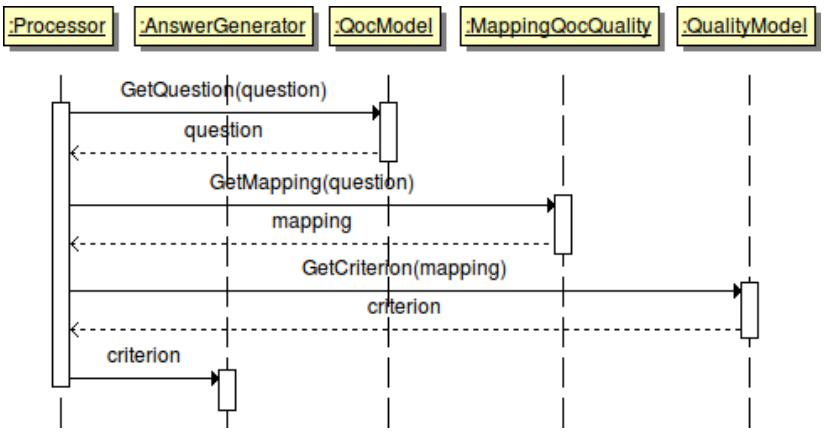


Figure 5.9: Sequence diagram for computing *Design Rationale* questions

5.4.3 Providing Support

Answering design rationale questions can be now done with the extracted criteria from the quality model. These criteria is the reason that justifies the design choice being asked in the question. Thus, a intuitive answer can follow the next grammar:

Because the ergonomic criterion + criterion.description

in case where only one criterion justifies the design decision, or:

Because the ergonomic criterion1 + criterion1.description + ... + criterionN + criterionN.description

as for instance in the example below:

Because the ergonomic criterion 'Items of any select list must be displayed either in alphabetical order or in any meaningful order for the user in the context of the task'.

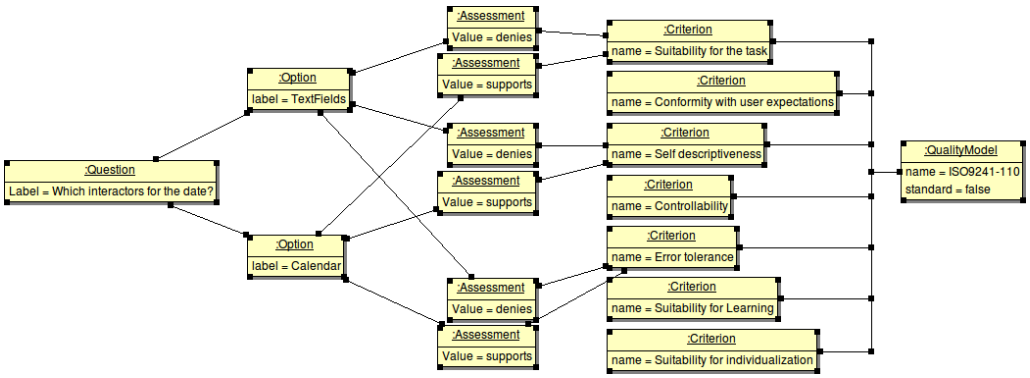


Figure 5.10: QOC and Quality models linked through quality criteria. Design rationale questions are directly retrieved from the QOC model. Answers are provided according to the criteria that support (Assessment) the selected option.

The next section provides an overview of the chapter.

5.5 Synthesis

In this chapter we have presented QUIMERA, a quality metamodel that unifies quality aspects from HCI and Software Engineering, setting the bases for a quality driven adaptation of UIs through quality models.

The chapter started with a review of the concept of design rationale, describing briefly the main approaches for design rationale notations, and making focus on a specific notation: the "Questions, Options and Criteria" design rationale notation, also known as QOC.

The chapter describes then QUIMERA, a quality meta-model to improve design rationale.

The meta-model is introduced by presenting the principles that guided its design, the quality perspectives and how the quality meta-model deals with such perspectives. Then, the chapter details the different elements of the meta-model, explaining its structure from the optic of Global quality and Local quality.

After this, the chapter provides two different examples of instantiations of QUIMERA. The first example is a quality model for HCI that is based on ergonomic criteria. The second example is a software engineering based quality model for code-source metrics. Then, a four

steps method is provided to describe how to instantiate the quality meta-model.

Later, the chapter provides an approach for evaluating design rationale alternatives through quality criteria. The approach is funded on the QOC design rationale notation and a quality model conforming to QUIMERA. The approach is illustrated through an example that shows how to evaluate design alternatives with a real quality model. The discussion continues with a discussion of some advantages and limitations of the approach.

The chapter ends with an explanation strategy for supporting design rationale questions at runtime, which is based on both a QOC model and a quality model based on our quality meta-model QUIMERA.

Next chapter presents the software contribution, which defines an architecture implementing all the presented concepts, provides the details of the implementation of the different explanation strategies, presents a running prototype based on these concepts and the results of an evaluation that we have carried out with real users.

6

Self-Explanatory UIs in Action: Implementation and Evaluation

> *Talk is cheap. Show me the code.*

Linus Torvalds,

RELATED PUBLICATIONS

1. García Frey, A., Calvary, G., Dupuy-Chessa, S., and Mandran, N. Model-based self-explanatory UIs for free, but are they valuable? In *Proceedings of the 14th IFIP TC13 Conference on Human-Computer Interaction (INTERACT'13), 2-6 September 2013, Cape Town, South Africa* (2013), Springer,
2. García Frey, A., Calvary, G., and Dupuy-Chessa, S. Users need your models! exploiting design models for explanations. In *Proceedings of HCI 2012, Human Computer Interaction, People and Computers XXVI, The 26th BCS HCI Group conference (Birmingham, UK)* (2012)
3. García Frey, A., Céret, E., Dupuy-Chessa, S., Calvary, G., and Gabillon, Y. Usicomp: an extensible model-driven composer. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems* (New York, NY, USA, 2012), EICS '12, ACM, pp. 263–268

This chapter firstly describes UsiExplain, a generic architecture for the development of self-explanatory user interfaces. This architecture respects the design principles that a model-based help system should follow, as presented earlier in chapter 4.

After presenting the generic architecture the chapter describes our specific implementation. For the purpose of the implementation, we have created UsiComp, an integrated and open framework that allows designers to create models and modify them at design time as well as at runtime.

Once the implementation details have been covered, the chapter introduces UsiCars, a running prototype entirely based on UsiExplain.

The prototype UsiCars has been used for evaluation purposes. This evaluation is then presented. It consists in a qualitative study carried on with twenty real users. The chapter ends discussing the study, describing its findings and conclusions.

6.1 UsiExplain: A Model-Based Generic Architecture

Figure 6.1 introduces the principles of the architecture of the self-explanatory user interface. The self-explanatory UI consists of a model-based help system or self-explanatory facility, plus the user interface of the target application. Both UIs can be mixed into one single user interface by weaving the UIs at different levels of abstraction as previously explained in chapter 4.

Both user interfaces are model-based, so they are both composed of the user interface models that are used to generate the user interface, plus the functional core, as depicted in the picture.

The user interface is generated by transformation according to a model-based approach. In the context of this research we have chosen the Cameleon Reference Framework[1]. The models and meta-models involved in the generation of the user interface are directly accessed by the self-explanatory facility when an explanation is requested by the user.

[1]For a description of the Cameleon Reference Framework, the different levels of abstraction of a user interface, and how a UI is generated from models representing these levels of abstraction, see chapter 3

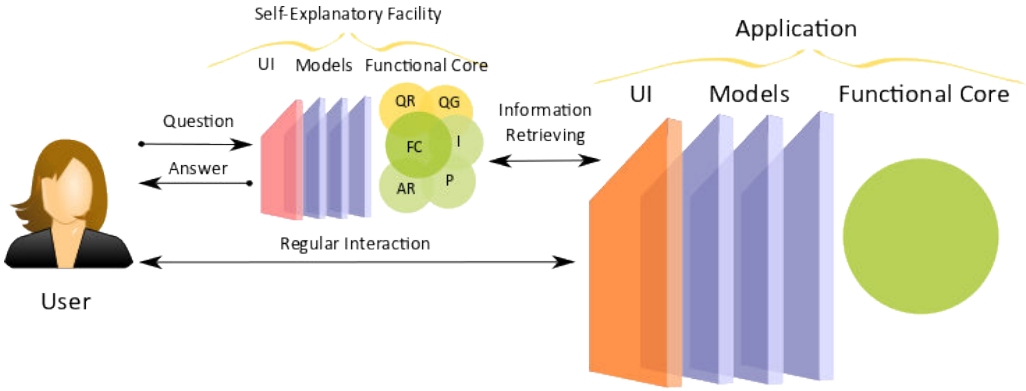


Figure 6.1: The Self-Explanatory User Interface consist of the UI of the application (right) plus the self-explanatory facility (middle) being both of them model-based UIs.

This procedure of accessing the underlying models is done in the functional core of the self-explanatory facility through five different modules. These modules in which this functional core is decomposed are:

The Questions Generator The questions generator (QG) is responsible for generating the list of questions that the system understand or is able to answer.

The Questions Renderer (QR) is responsible for the presentation of the list of questions.

The Interpreter The module Interpreter (I) is responsible for analysing the users' request, inferring the type of question and its different parameters if there is any.

The Processor This module (P) computes the answer or explanation based on the type of question and the parameters that have been determined by the Interpreter.

The Answers Renderer (AR) is responsible for the presentation of the answer back to the user, in an understandable way.

Each of these five modules of the self-explanatory facility has full access to the models of the underlying application at runtime.

Figure 6.2 provides a different overview of the architecture, including the details of the accessed models and meta-models. In this figure, both UIs are combined into a single UI (*Weaved UI*) on which the interaction with the user happens. This interaction is managed by a *Controller*. The *Controller* links the application logic from the functional core to the

UI and vice versa. For instance, it is in charge of performing navigational operations (for example, navigating between different windows of the same UI or accessing a different page of the same website), and linking the functional core with the UI for computational purposes (for instance, when the application requires to save the document to a file or loading an existent resource).

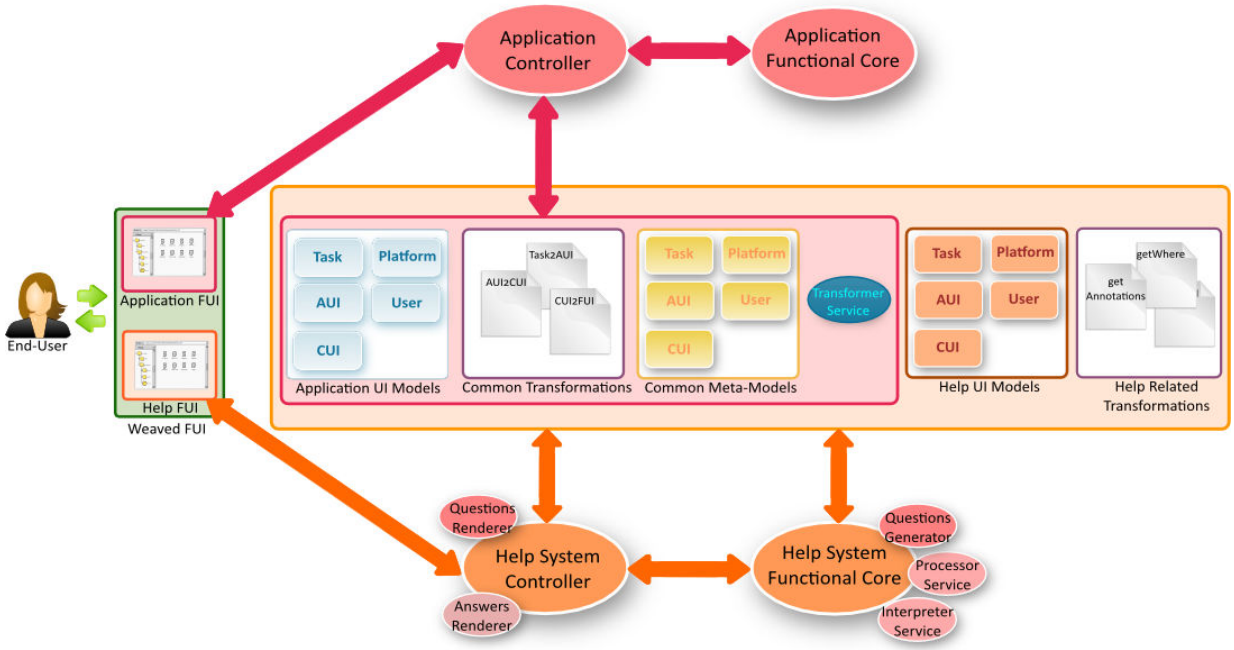


Figure 6.2: Generic architecture for model-based self-explanatory help systems. The functional core of the help UI accesses any (meta-)model at runtime.

In this architecture, the UI of the application as well as the UI of the help system have their own *Controller*. The three vertical arrows in figure 6.2 represent the access to the different model-related elements. For instance, the *Controller* of the application can access to the models, meta-models, and transformations of the user interface of the application in order to generate the UI. In the same way, the *Controller* of the help system can also access the models, meta-models, and transformations of the UI of the help system for generation purposes. The functional core of the help system can access to any model-related element of both application UI and help UI, in order to find those elements that are necessary to compute the

requested explanation.

From the end user's point of view there is only one weaved UI. The *question renderer* is in charge of providing the user with a mechanism for asking questions. This can be by entering the question in natural language or by selecting the desired one from a list of questions. When the user requests support, the *help controller* receives the request and passes it to the *interpreter* in charge of understanding the question. This interpreter can, for instance, parse the natural language input of the user or even recognise the gesture triggering the question with a gesture recognition system. The interpreter says to the *processor* what support information needs to be computed such as the type of the question and its parameters. The *processor* computes such information by accessing the models at runtime, according to the explanation strategy that has been specified for such type of question. This can be done by applying special help transformations that query the models at runtime, or by accessing the models via a special API as explained later. The *processor* can query all the models independently if they belong to the application or the help system, and using exactly the same *help transformations*. This is possible because all the models conform to the same meta-models. Once the information has been retrieved from models and computed by the *processor*, it is prepared for the end user by the *answers renderer*. The *answers renderer* can update the UI with the desired information so the user can use it. The *answers renderer* is then responsible for managing how the information is presented, for instance, in some text or voice using natural language, or with an animation of the mouse cursor showing some procedure.

UsiExplain covers all the types of questions and their related explanation strategies that have been presented in previous chapters, i.e.:

Procedural questions for answering *How* question.

Purpose that provides feedback about *What is it for* questions.

Localization that replies to *Where* questions.

Availability answering the question *What can I do now*.

Behavioural questions explaining *Why I can't* perform a task.

Design Rationale that answers questions about the design rationale of the UI.

The UsiExplain architecture is prepared to extend this set of generic questions with new types, according to the principles and the design of explanation strategies introduced in previous chapters.

To ease the implementation of such architecture, we have develop UsiComp [50]. UsiComp is integrated and open framework that allows designers to create and modify models at design time as well as at runtime. The next section describes this framework in detail.

6.2 UsiComp: a Services Oriented Framework

In the context of the UsiXML project, we, the HCI group, have created UsiComp. UsiComp is an open framework for creating models and simplifying the creation of user interfaces through transformations. UsiComp relies on a service-based architecture. It offers two modules. A design module for creating and editing models through an integrated tool, and an execution module for managing the runtime. The implementation has been made using OSGi services (Open Services Gateway Initiative, [86]) offering dynamic possibilities for using and extending the tool. The next section describes these concepts.

6.2.1 Services and OSGi

The term service refers [157] to:

> A set of related software functionalities that can be reused for different purposes, together with the policies that should control its usage.

OSGi is a modular Java framework that allows modules or subsystems, known as bundles, to be dynamically added and removed from a running Java Virtual Machine (JVM). As OSGi is layered on top of a JVM it continues to permit access to all the native features of the underlying JVM as well as allowing incorporation of native non-Java code via the JNI (Java Native Interface) framework to an OSGi based application (see figure[2] 6.3)

[2]Image by Michael Grammling publicly available at `http://en.wikipedia.org/wiki/File:Osgi_`

The choice of OSGi makes it easier to incorporate a multitude of different devices, permits the modification of the self-explanation related modules at runtime, and eases the property of distributabitility of the self-explanatory user interface across different platforms at runtime.

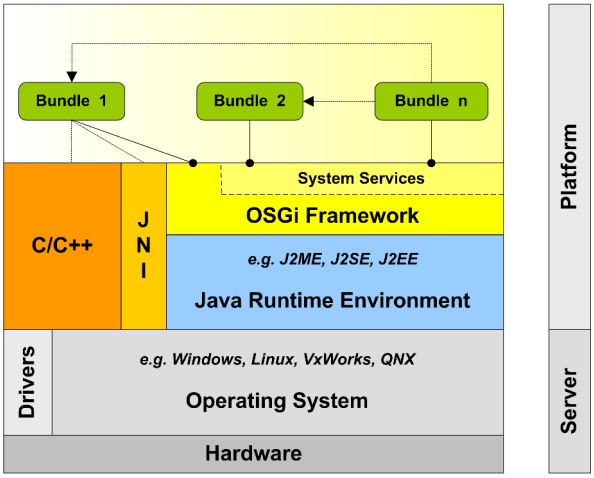


Figure 6.3: Overview of the OSGi Layers

As illustrated in figure 6.3, at the bottom of the OSGi layering is the Operating System and Hardware, which may be a standard PC Desktop operating system such as Windows or any Linux based distribution, as well as mobile devices such as smartphones or tablets. A Java Runtime Environment (JRE), which is composed of a JVM and a collection of classes that implement the Java API for the underlying device operating system and hardware, runs alongside native applications that may be written in C/C++ or another language. The JRE can be one of many environments such as Java ME (Micro Edition) for embedded devices, Java SE (Standard Edition) for desktop platforms and Java EE (Enterprise Edition) for server platforms. Each platform provides a different set of services which can be exploited by applications using the same OSGi framework. The implementation of the UsiComp framework uses the Java SE desktop edition.

There are several implementations of the OSGi Framework; UsiComp is based on the Equinox[3] implementation developed for the Eclipse project. However, this could be replaced

`layer.png` under the Creative Commons Attribution ShareAlike 3.0 license.

[3] Eclipse Foundation. Equinox OSGi Release 4 (Equinox), 2009.

by any other implementation conforming to the OSGi standard such as the Knopflerfish[4] implementation, or the Apache Felix[5] package. We have chosen Equinox to ensure a full integration with other necessary eclipse related technologies used in the development, such as Ecore in which the meta-models are described.

Bundles in OSGi are regular JAR (Java ARchive) files with an additional bundle.manifest file which specifies that bundle's dependencies (in terms of other packages required for operation) as well as which packages it provides to the OSGi framework which can be used by other bundles. The bundle manifest specifies an "Activator" class which is called when the bundle is loaded and unloaded from the framework. This Activator class is responsible for starting and stopping any services provided by the bundle as well as obtaining references to services it requires.

Based on OSGi services, we have developed two different modules that compose the UsiComp architecture. An overview of this architecture is provided in the next section.

6.2.2 UsiComp Overview

UsiComp is composed of two different modules as shown in figure 6.4), the design module (at the top of the figure), and the runtime module (at the bottom). Both modules share common resources: meta-models, models and transformations. This section describes both modules in detail, starting with the design module, then providing a brief discussion about the common resources, and finally describing the runtime module and the code generation.

6.2.2.1 Design Module

The design module includes a visual editor (figure 6.5) for designing and prototyping UIs. The UsiComp editor offers the following functionalities.

First, it allows designers to define all the models and transformations needed to produce a UI. The UI of the UsiComp editor is divided into three different areas (figure 6.5): 1) a toolbar

[4]Makewave AB. Knopflerfish OSGi Release 4 (Knopflerfish 2), 2007.
[5]Apache Software Foundation. Apache Felix OSGi Release 4 (Felix 2), 2009.

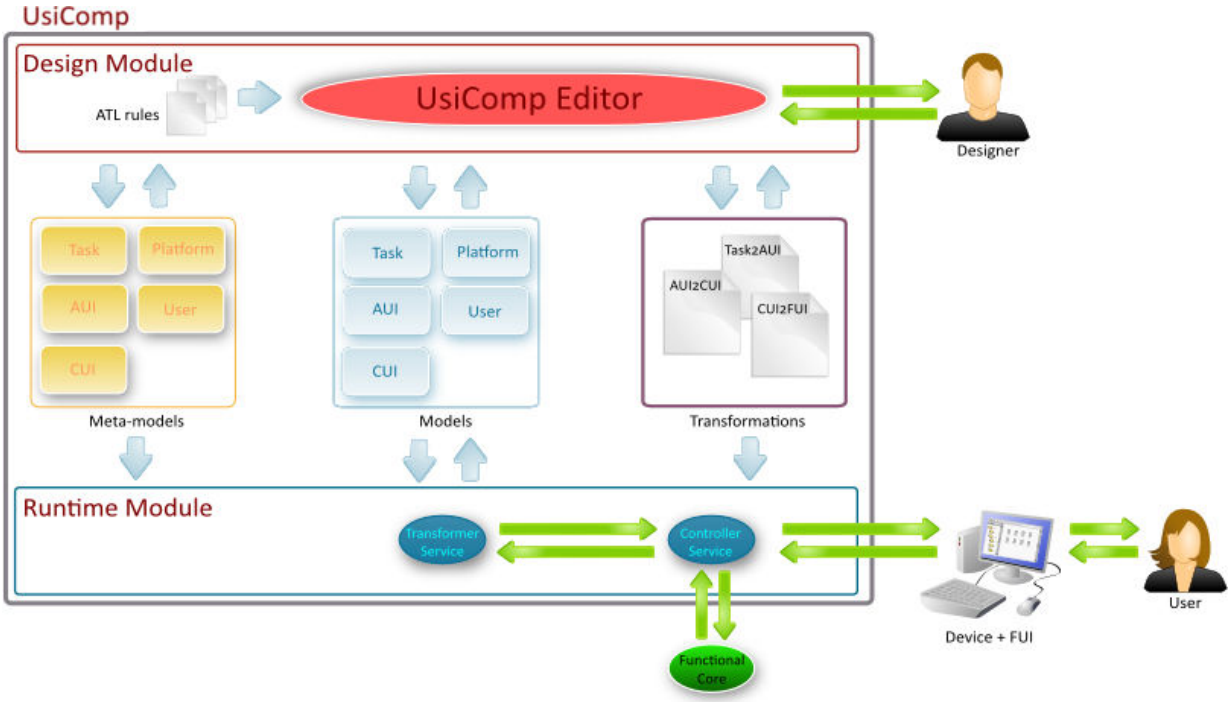


Figure 6.4: UsiComp software architecture: meta-models, models and transformations at the heart of both design time (IDE for designers) and runtime (FUIs for end-users).

with the most common actions, 2) the workspace presenting graphical representations of the models, and 3) the right panel which provides access to the different elements of each meta-model. Designers can create models by picking up the needed components and combining them. For instance, figure 6.5 shows the UsiComp editor and three models with their respective transformations. The model at the top of the figure is a task model, represented with the CTT notation. This task model is transformed into an AUI model represented with blue boxes. These blue boxes show different Abstract Interaction Units and their arrangement. The AUI model is in turn transformed into a graphical CUI model that UsiComp represents with a mock-up.

Transformations between models are composed of rules. A rule specifies how one specific set of elements of a source model is transformed into a set of target model elements. Designers can select what rules they want to apply to a given model, and the system will auto-

matically compose the resulting transformation. These rules are represented by arrows from the source element to the target. Some common rules are already available in the system (for instance, transform a AUI unit into CUI widgets such buttons, checkboxes, etc), but designers are free to add other rules if needed. As previously stated, transformations and rules are written in ATL.

The UsiComp editor verifies that the designed models comply with their corresponding meta-models. For instance, a binary operator in the task model must link two different tasks. This is done through the validation facilities provided by the EMF tools.

The UsiComp editor also composes and compiles the transformations and rules thanks to an integrated ATL compiler.

The resulting Final UI, which is the code of the UI, can be directly executed from the IDE (green play button on the toolbar) giving designers the opportunity to preview the generated UI.

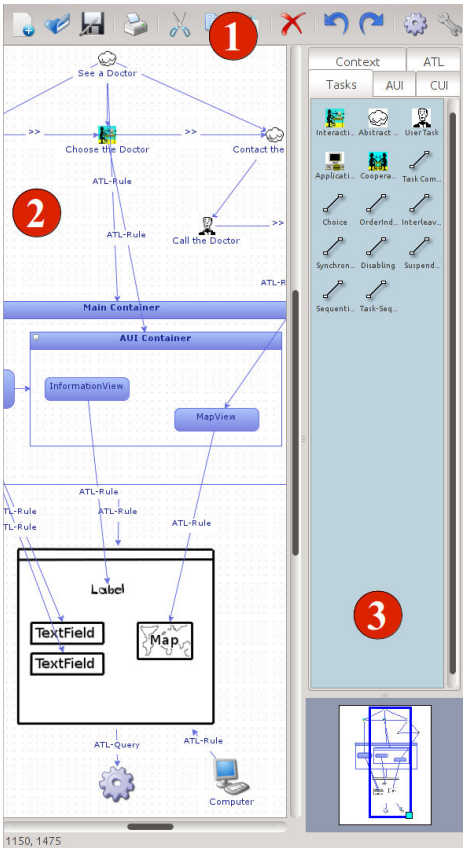


Figure 6.5: UsiComp Development Environment. From Top to Bottom: Task model, AUI model, CUI model. Transformations are represented by arrows.

6.2.2.2 Meta-Models

For this research, we use adapted versions of different UsiXML models. Illustrations and explanations of such meta-models are provided in appendix B. All the meta-models have been implemented using the Eclipse Modeling Framework (EMF). EMF [140] is a modelling framework and code generation facility for building tools and other applications based

on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. We have implemented all the necessary meta-models and models according to the Ecore EMF format.

The models conforming to these meta-models are transformed from one to another by transformations, that are described in the next section.

6.2.2.3 Transformations

The meta-models have been used not only for instantiating models but also for defining transformations between these models. For these transformations we have chosen ATL, the Atlas Transformation Language [66]. ATL is the ATLAS INRIA and LINA research group's answer to the OMG MOF/QVT RFP. It is a model transformation language specified as both a meta-model and a textual concrete syntax. In the field of Model-Driven Engineering (MDE), ATL provides developers with a mean to specify the way to produce a number of target models from a set of source models.

The ATL language is a hybrid of declarative and imperative programming. The preferred style of transformation writing is the declarative one: it enables to simply express mappings between the source and target model elements. However, ATL also provides imperative constructs in order to ease the specification of mappings that can hardy be expressed declaratively.

An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. Besides basic model transformations, ATL defines an additional model querying facility that enables to specify requests onto models. ATL also allows code factorization through the definition of ATL libraries.

ATL transformations are used in both UsiComp modules, either at the design time or runtime. These modules are explained next.

6.2.2.4 Runtime Module

UsiComp is composed of several services. This section describes only those that are relevant for self-explanatory user interfaces. These services are the Controller and the Transformer.

The main service is the Controller Service (figure 6.4). The Controller Service is in charge of orchestrating the whole process in which a UI is generated by successive transformations. Transformations may be reifications or abstractions. Currently, only reifications have been implemented and integrated into UsiComp. However, the architecture is fully generic, and so capable of integrating abstractions as well.

The Transformer Service (Figure 6.4) is a generic transformation service that can apply any transformation to any model or models, producing as a result models (in the case of a model-to-model transformation) or text (in the case of a model-to-text transformation). The Transformer service relies on a set of meta-models that the transformations and the models must conform with. Next section provides a brief description of these meta-models.

The functionality of the runtime module is described as follows:

- The runtime module is running on the server side listening for incoming connections.
- Once a new device becomes available to the framework (a specific client is installed into the device for this purpose), UsiComp identifies its specific platform model containing the platform details. The current version of UsiComp contains platform models specified by hand.
- To produce a UI for the new client, the Controller Service manages the transformations, their order of execution and their related models and meta-models, calling to the Transformer Service as many times as needed. The platform model is considered in the transformation process to produce an adapted UI.
- In the transformation process, the Controller weaves the functional core of the application into the UI, embedding the calls from and to the UI.

The models, meta-models and transformations involved in the generation are directly accessed by the Controller Service, which is also responsible of linking the application logic from the functional core to the UI and vice-versa.

The development environment can be launched as a normal Desktop application or as a Web application embedded in an applet. Thanks to the OSGi services, it is possible to dynamically update the editor without stopping the application. For instance, updating a service or replacing the transformation language for another one can be dynamically achieved.

6.2.2.5 Code Generation

UsiComp currently supports the generation of Java code. The Java code is directly generated from CUI models with a special ATL transformation from model-to-code. ATL does not only support model to model transformations, but also model to "primitive value" transformations. This last type of transformations is called queries. They can be used to generate text from models. In this particular case, the primitive value is a String data type containing all the generated code of the UI.

The code generation is directly done by transformation instead of using external tools for several reasons. First, most of the technologies that already exist focus on one language only (as for instance JaMoPP [56] for Java), or only one programming paradigm, mainly imperative in most of the cases. As the generated UI must be platform independent, the code generation cannot rely on only one specific language or paradigm. For instance, we would like to generate GTK UIs in the future for a functional language such as Haskell. Not all the languages and paradigms are supported by external generators, so integrate an external tool each time is not always possible.

Technically, the code generation is done by parsing the CUI model with a Depth First Search algorithm, i.e., translating the first element of the CUI model (at the top of the model, for instance, the main window) and exploring/transforming as far as possible along each branch before backtracking. This is possible because the CUI meta-model forces a free loops tree-like CUI models.

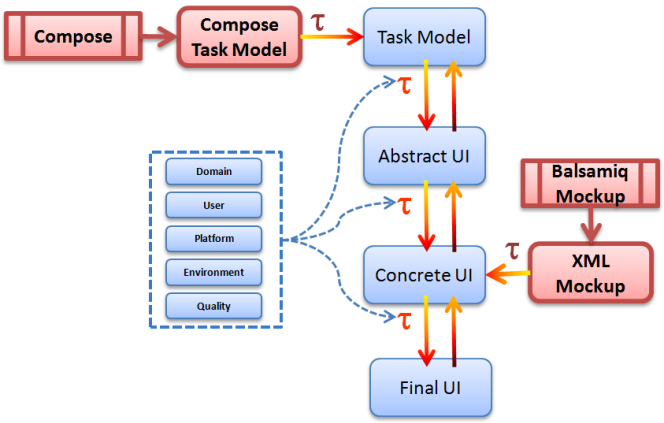


Figure 6.6: Two examples of the UsiComp extensibility. A task model is generated from an external tool called Compose. A CUI model can also be generated from a mockup instead of transforming the AUI model.

6.2.2.6 Extension abilities

UsiComp has been developed in common with another PhD student as it is not limited to self-explanation purposes but also to enrich the UI development process allowing designers to generate the UI with different models to those specified by the classical Cameleon approach. This ability to provide extensions is also a research area in our research group. The extensibility feature of UsiComp is studied in Eric Céret's PhD. Interested readers can refer to [21, 50] for more information. We show here only two examples for illustration. These examples are summarized in figure 6.6. In this figure, the classical Cameleon transformation sequence and its related models and transformations are extended at different levels. The first extension is done at the task level, where the task model is not provided by the designer but generated from an external tool called Compose. The second one is done at the CUI level, where the CUI model is obtained by a Balsamic Mockup instead from the classical AUI level.

6.3 Relationship between UsiExplain and UsiComp

Figure 6.7 shows the relationship between UsiExplain, the generic architecture for self-explanatory UIs, and the framework UsiComp. As readers may notice by comparing the figures 6.2 and 6.4,

the *Controller* from the runtime module of UsiComp is implemented through two controllers in UsiExplain as explained before, one for the help system, and one for the target application. In the same way, the *functional core* in the figure corresponding to the architecture of UsiComp 6.4 is implemented in UsiExplain through two different functional cores, again one for the help system, and the other one for the application.

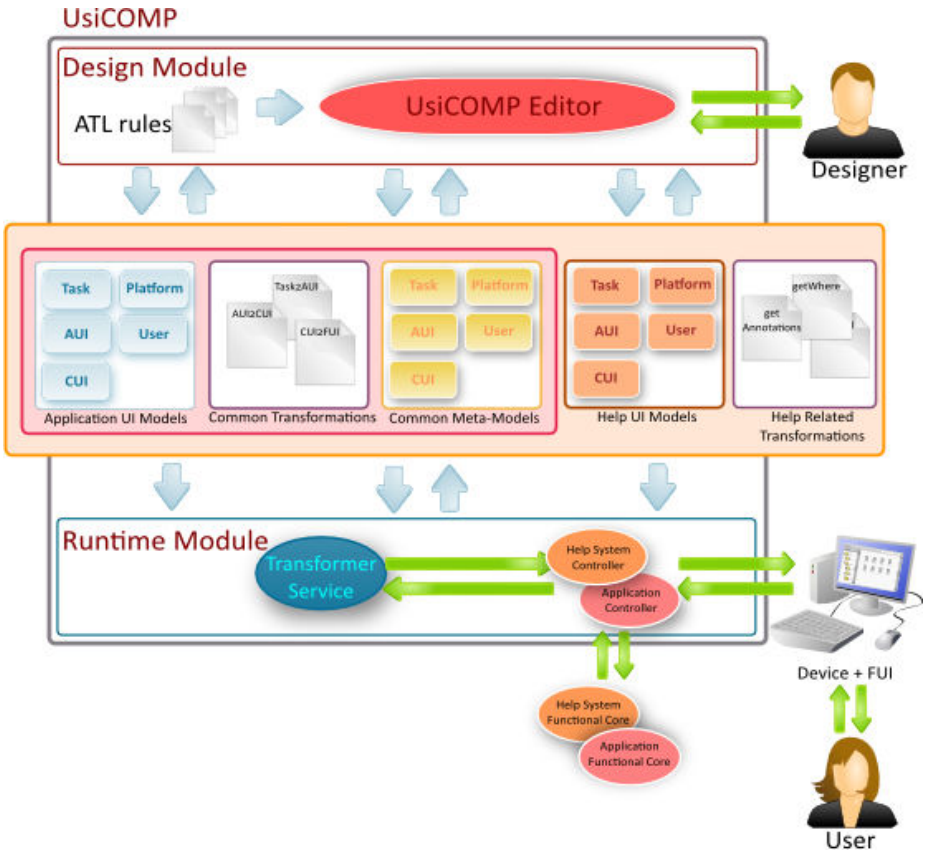


Figure 6.7: Relationship between the UsiExplain generic architecture and the UsiComp framework.

In both cases, the *Controllers* access to models and meta-models at runtime for generating the UIs. The difference here between both infrastructures is that the *functional core* of the help system has direct access to such models and meta-models at runtime too, to guarantee that answers can be composed based on these elements at runtime.

The presented architecture follows the MVC architectural pattern, where the “Models” in

MVC are all the elements represented in the middle of the figure (models, meta-models, and transformations), the view is the UI used by the user to interact with the application, and updated by the controller, and the controller keeps the link between the "Model" from MVC and the "View" or UI in our case.

Moreover, the UsiExplain generic architecture provides designers and developers with:

- a questions generator for computing generic questions.
- an answers generator with a generic coverage of questions.
- a complete set of explanation strategies for computing six different types of generic questions and its related answers.

The six generic question types supported by the explanation strategies provided with the architecture are:

Procedural questions for answering *How* question.

Purpose that provides feedback about *What is it for* questions.

Localization that replies to *Where* questions.

Availability answering the question *What can I do now*.

Behavioural questions explaining *Why I can't* perform a task.

Design Rationale that answers questions about the design rationale of the UI.

As previously presented, other different questions and answers can be integrated as well in the architecture by adding new explanation strategies.

The next section describes a prototype entirely based on UsiExplain.

6.4 UsiCars: an UsiExplain Based Prototype

This section describes UsiCars, a prototype that relies on UsiExplain for supporting users at runtime. This prototype has been created for evaluation purposes, described later in the chapter.

The section starts with a description of the prototype. Second, it describes the dialogue that the prototype has used to allow users to request for explanations, discussing some considerations of the current coverage of the questions in the prototype with regard to the UsiExplain infrastructure.

6.4.1 Prototype Description

The prototype consists in a cars shopping website called UsiCars. This website is inspired by a real site from a real car manufacturer. We have reproduced only the part of the website that is devoted to the selection and configuration of the vehicles, keeping the options and the structure of the original website.

This website was chosen for two main reasons. The first one is that we needed to use an interface that contains knowledge that is understandable and accessible by all the participants, but complex enough for not being easy to use. A website for configuring cars covered this point as all the participants understand many of the car related concepts, but at the same time there are enough specific options with domain related concepts to create complex tasks that are non trivial to perform. The second reason is that we found the original website difficult to use by real users in different forums.

The reproduction of the website was done by a reverse engineering process. The first step was to explore all the different tasks that the user can perform to select and configure the a vehicle. We created a task model according to this information. Secondly, we created a transformation to obtain an Abstract UI model that conforms to the structure of the original website. Thirdly, we wrote another transformation to generate the Concrete UI model from the Abstract UI model. This transformation produces all the widgets that we find in the original website. We also used the same images and we respected the same sizes for all the widgets from the original site, to ensure that we obtain the same usability properties. Finally, we wrote another transformation to generate the Java code and produce the resulting site.

In each of the model to model transformation, we generated not only the target model but also mapping models that keep track of the successive transformations of an element

from one model to another. For instance, in the transformation from the task model to the Abstract UI model we generated a Mapping-Task2AUI model that specifies what tasks are transformed into what Abstract UI elements. The same principle was applied to obtain a Mapping-AUI2CUI model. This allow us to go through the transformation chain and, for instance, retrieve the source task from which a button has been generated.

Figure 6.8 shows an excerpt of the UI of the prototype. The UI is divided into two main areas. A big main area in the middle and a thin area at the bottom. The main area of the UI has two different roles. On the one hand, it serves as a visual feedback for the user when he/she selects a car model or changes the colour of the vehicle (figure 6.8). On the other hand, it can show dialogues containing all the possible options that the user can select to configure the car with. The thin area at the bottom allows users to navigate through several categories of options for accessing different features of the car such as the electronic equipment or the external colour of the vehicle (figure 6.9).

The prototype was build according to the UsiExplain architecture. The infrastructure con-

Figure 6.8: Screenshot of the prototype. Choosing the model.

sists of two model-based UIs, the self-explanatory facility for providing the help, and the UI of the target application for configuring a vehicle.

6.4.2 Self-explanatory dialogue

The questions were presented in a textual form inside a dialogue (figure 6.10). Textual answers showed up after clicking on the desired question. In the experiment, questions were presented one by one and only at the end all the questions were shown together. We did not filter out any question in this dialogue, i.e., all the possible questions that the system was able to answer were proposed to the users. The reason for this was to show the users all the questions, so they can better realize if the self-explanatory system could cover their expectations for the given type of question. For instance, if they realize that their question is not covered by the system because it is missing in the list.

The next section describes the experiment that we have conducted based on this prototype.

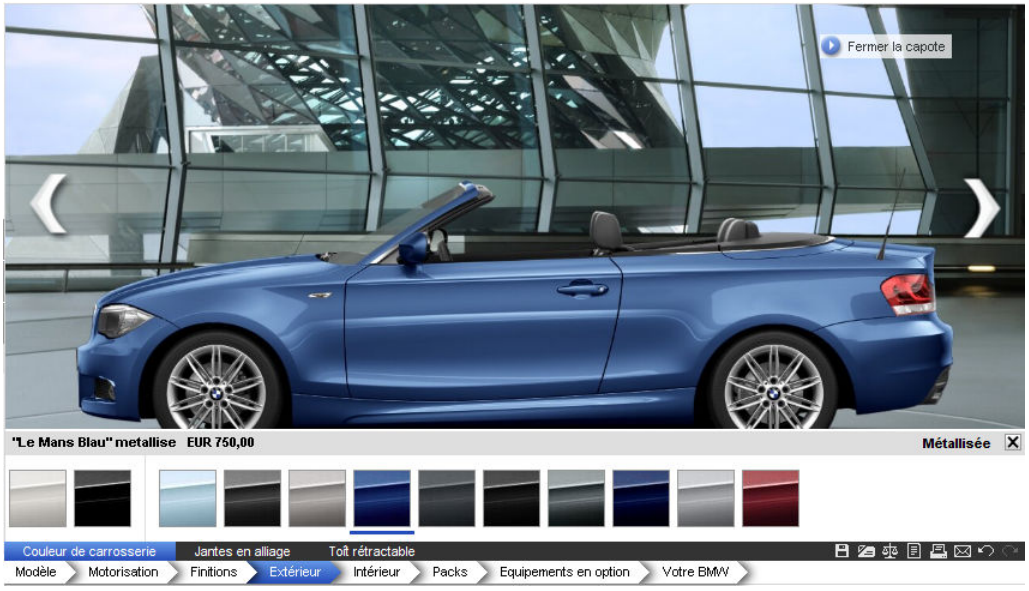


Figure 6.9: Screenshot of the prototype. Selecting the external colour.

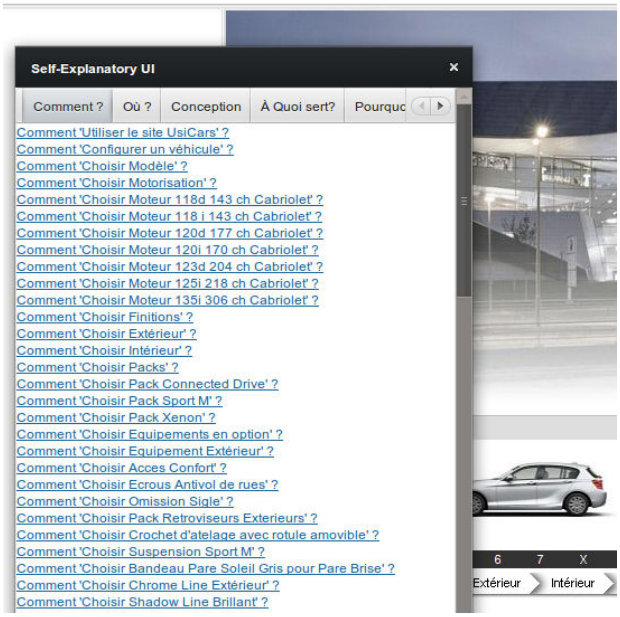


Figure 6.10: Self-Explanatory dialogue showing the full list of types and questions.

6.5 Evaluation

We conducted an experiment to evaluate the possible added value of the previous model-based self-explanations. This section starts describing the participants involved in such experiment, and then, it describes each of the different phases that integrate the evaluation protocol.

6.5.1 Participants

We selected 20 participants, all between 23 and 39 with an average age of 27.4. From the 20 participants, 12 were male and 8 female. We recruited individuals regardless their experience with interactive systems because the possible added value of model-based explanations can vary regarding the experience of each profile.

6.5.2 Evaluation Protocol

To carry out the study, we broken-down the evaluation protocol into three different phases. These three phases were performed in order for all the participants.

1. In the first phase, we asked the participants to answer a questionnaire. This questionnaire allowed us to better known the background of participants, to understand their habits regarding how they use new technologies in general, what are their common uses, the kind of applications they use with a relevant frequency, the problems they use to find with these or other applications, as well as their habits for solving these problems. The questionnaire also included questions regarding how participants used the help provided by the applications they use, and how they used to proceed in case they have a problem with the application. A software recorder was used to record all the answers of all the participants.

2. In the second phase of the experiment, we asked the participants to use the prototype that we had developed to this aim. We asked them to complete 10 different tasks in an established order. All the participants received the identical 10 tasks. We randomized the order of the tasks for each participant to avoid side effects such as the influence between different tasks or memory effects that can help users to accomplish the tasks better in a certain order. This part of the experiment was conducted on a laptop and the audio was recorded. We asked the participants to verbalize their thoughts, specially the questions they would like to ask to the system and the problems that they find when accomplishing the tasks.

3. The third part of the experiment presented the prototype including a self-explanatory dialogue that contained one type of question at a time. The six questions discussed previously were presented one after another again in a randomized order. For each type of question, the dialogue showed all the possible questions that the participants could ask. Every time we showed a new type of question, we asked the participants their opinion about it, including the possible advantages and disadvantages of asking

that question to the UI. We asked as well if the given type of question could be useful in the previous phase of the experiment. At the end of the third phase, all the types of questions were shown together into the same self-explanatory dialogue, and we asked some more general questions that are discussed in section 5.

The next section deeps into the second phase of the experiment, providing more detailed information about the tasks involved in this experiment.

6.5.3 Tasks

The motivation for the second phase of the experiment was to confront the users with different kind of problems that are frequently found in UIs. To this end we designed 10 different tasks. The tasks were selected according to their complexity, ranging from easy tasks to more complex ones. We did not force any specific problem in the tasks that could be easily solved by one of the previous questions. Instead, we tried to reproduce a realistic use case with a varied set of tasks so the answers of the participants in the phase 3 were not influenced by the second phase.

The 10 different tasks that we asked the participants to complete are shown in the table 6.1. The accomplishment ratio indicates whether the participants were able to complete the tasks at all. A few users that got stuck and required hints were counted as unsuccessful. The accomplishment ratio gives an idea of how difficult each task was, regardless the expertise of the user.

Some of the tasks involved selection with searches through small lists (1, 2, 4, 8) while others involved selection through lists having multiple options and categories (5, 7, 9) in different locations. Tasks 1, 2 and 4 involved selections through images while the rest of the selection tasks were through options in textual form. Other tasks involved verification (6, 7, 10), comparison (7), or manipulate cars related terminology that was more or less easy to understand (1, 4, 6, 8).

We used the accomplishment ratio in the last part of the experiment, specially when we

Task Description	Accomplishment ratio
Select a *Cabriolet* model	20/20
Select a diesel engine for less than 35.000 €	17/20
Choose a sport finishing touch	15/20
Change the external colour to Le Mans Blau	20/20
Ensure that the model has a navigation system. If not, add one	12/20
Ensure that the model has a Terra leather upholstery. If not, choose blue leather instead	12/20
Make sure that you can listen music in the car. If not, choose the best audio system available	12/20
Select the Connected Drive pack	18/20
Select a Maintenance Contract of your choice	10/20
Visualize the result and check that everything is OK. If not, try to solve the problem	12/20

Table 6.1: List of tasks and their accomplishment ratios. The tasks were randomized to avoid side effects such as the influence between tasks or memory related effects. The accomplishment ratio give an idea of the difficulty of the task.

asked the participants if they believed that the model-based explanations could help them to complete one of the problematic tasks, or doing it in a more efficiency way. The next section discusses the results of the qualitative analysis that we carried out with all the collected data.

6.6 Qualitative analysis

A large amount of qualitative data was collected from the experiment. We extracted around three hundred comments from the records made during the second phase of the experiment, the one in which participants were asked to complete the list of tasks. The selected method of analysis was the thematic type [112], This method is focused on the answers and comments recorded during the experiment, and classified into categories later. The aim of the thematic analysis is to group together answers or parts of answers that have the same meaning. The thematic groups were then analysed to identify the different categories of opinion. The objective is to gather and list all the themes covered by the answers to reflect the widest possible range of opinions, distinguishing the positive ones from the negatives.

From the extracted comments, we identified around 250 verbatims that referenced types

Question type	Example Verbatims	Occurrences
How	I don't see how to do it	13
Why	Why I need to register?	21
Where	And where do I find the maintenance contracts	119
What is it for	I'm browsing the tabs to see what they do	7
What can I do now	I must find my way (inspecting all the UI with the mouse)	2
Design rationale	Why they are not ordered by type?	1
Other types	What does Cabriolet stand for? *(Definition)* What are the differences between the packs? *(Differences)* Is it included in the price I guess? *(Confirmation)* What happens if I click here? *(What if)*	81

Table 6.2: Relationship between question types and occurrences extracted from the records during the second phase of the experiment. An example of verbatim illustrates each type.

of questions either in an explicit or implicit way. Only those verbatims that clearly related a question type where considered. For instance, verbatims like "I don't know where the contracts are" were classified as an implicit question of type Where. The table 6.2 shows the results of this classification, as well as some illustrative verbatims. It is significant that most of the verbatims addressed navigational problems (where + how) mainly due to usability issues and to the nature of the tasks (table 6.1). The high number of 'Other types' is mainly due to questions about semantic information relating concepts specifics of the domain. These questions are described in section 6.2, while next section presents the findings for both positive and negative opinions, as well as some revealed limitations of the approach.

6.6.1 Findings

In the first phase of the study we collected the data described in the previous Participants section, and we also found that 16/20 liked new technologies, 17/20 use new technology everyday, and 20/20 have found problems in their use. To face these problems, 11/20 inspect the UI to try to solve it by themselves, 8/20 ask other people about the problem, 15/20 search for solutions in the Internet, and 7/20 use the help provided by the system.

The last phase of the study revealed that questions of types How and Where were identi-

fied by most of the users (15/20) as useful and helpful with statements such as “it can be very useful in certain situations” or “It could be very helpful for locating all the options of the vehicle in a faster way”. This last statement refers also to a gain of time, which was also identified as a positive value by a total of 10/20 users with statements such as “It is a gain of time” or “it makes me go faster without losing my time”. The good acceptance of How questions contrasts however with the low number of verbatims. This suggests that users find the information useful but they are not thinking of asking it. The help UI could encourage/propose questions in these situations.

The What is it for and Why questions were also identified as useful by an important number of participants, but less useful than the previous ones. This was mainly due to the fact that subjects did not find useful to ask for the purpose of some elements of the UI, such as checkboxes or labels, that already contain clear information about what they are currently doing. In the case of Why questions, the results did not showed a good acceptance by the participants as in the results found by [100, 80]. This was due to the fact that the questions proposed by our algorithms did not cover all the possible range of questions that the participants asked. For instance, as our algorithms rely entirely on the task model, our system could not answer why questions concerning the functional core of the application such as Why there is no diesel engines? (for specific kinds of Cabriolet cars).

Finally, the What can I do now and design rationale related questions were found to be not very helpful by most of the participants (16/20), according to statements such as “I don’t see where I would like to ask this question” (for what can I do now?) or “I am not interested in this information, all I want is to buy my car” (for design rationale questions). In case of design rationale questions, this result is due to the fact that the questions proposed in the prototype were probably simple questions that should be reconsidered with real designers in order to propose more relevant questions.

At the end of the third phase, when we presented the help UI with all the types of questions together, the study revealed that in general, model-based self-explanatory facilities were identified as “useful” and “helpful” by most of the participants (16/20). The study also re-

vealed question types that were unsupported by our current implemention. The analysis of the collected data suggest that our model-based self-explanatory UI, with minor design enhancements for major usability improvements, could have the potential to easily help the users in real-time applications. Next section discusses the possible model-based implications for the types of questions that were unsupported. Then, we discuss the usability suggestions extracted from the data for our particular implementation of the self-explanatory UI.

6.6.2 Unsupported types of questions

We identified other types of questions not explicitly supported by our system. A minor number of them referred to What if questions. Even if most of these verbatims come from users that showed a trial and error approach to understand the consequences of their actions in the UI (i.e., they don't know the consequences of an action but they perform such action on the UI anyway to see what happens), 2 users out of 20 did not use options from the UI because they did not know their possible side-effects. For instance, subject 9 did not perform one of the tasks because "I have fear of losing all the options". Supporting What if questions can help this minority of users to feel more comfortable with the UI. These kind of questions can probably be answered by analysing the operators of the task model and how they are transformed to CUI elements, (what elements of the CUI model become active/inactive as we enable/disable new tasks. These answers will probably require some improvements for side effects related to the functional core of the application (external to the UI).

We also identified a high number of verbatims requesting confirmation and validation from the UI. For instance, "does the car already have a navigation system?", "are the options included in the price?", were recurrent expressions used by the participants. This observation suggests that the feedback provided by the site was not enough for the users. Supporting questions about confirming and validating the user actions can help to overcome this usability issue. This may require new models for handling user actions, specially those that have effects beyond the UI.

A third group of questions not supported by the self-explanatory dialogue concerns def-

initions. Most of these questions were about specific car-related terminology and concepts such “What is the Tuner DAB?” or “What does Cabriolet stand for?”. To support these questions, the proposed model-based approach needs to be extended with semantic information, either by adding new models or by connecting the UI with sources of semantic information (internet).

Semantic information may be also necessary for answering questions about differences that we identified in a minor number of verbatims, for instance, What is the difference between the packs? (or eventually similarities).

6.6.3 Usability Suggestions and Improvements

We were also interested in usability observations. During the third phase of the experiment, where participants were confronted to the self-explanatory dialogue, 14 out of 20 suggested that they would like to type the whole question directly instead of clicking on a predefined answer inside a list. 13 out of 20 would like to access questions by typing keywords in a text area, and 4 proposed to use a vocal interface instead. These observations sustain some of the design principles for help systems of the literature, in particular, “Help should be accurate, complete and consistent” ([31, 135]), and “Help should not display irrelevant information” ([59]).

6 participants suggested to classify questions not only by question types but following the categories of the underlying site, for instance, grouping them by equipment or car models.

Regarding the answers, some participants argued that they don’t like to read explanations, specially those that have a significant length. With the models used in this approach, the information given in the answers can be represented in non textual forms. For instance, as the CUI model can store the screen coordinates of the widget, Where questions can be answered by highlighting the region of interest (as currently done in mac systems), and procedural questions can be explained by means of animations of the cursor over the widget coordinates.

Finally, some participants proposed that it would be preferable to use the questions not as a means to know how to find a specific option, but to “get there”. This suggests that self-

explanatory UIs could be used as software agents [35] to overcome the usability issues of a UI not only by explaining to the user how to solve the issue, but solving it directly if possible. For instance, navigating to the desired website instead of explaining what website the user should navigate to. This observation opens new research questions: can self-explanatory UIs benefit from agents? If so, what other models are needed and how this can be done?

6.6.4 Limitations of the experiment

The results obtained in the experiment are only representative for the ages of the registered participants, i.e., between 23 and 39 years old with an average age of 27.4. Further research is needed to understand if the sampling led to a bias in the study.

Design rationale questions were also shown to need improvement. Probably a further research with real designers with help to identify more design rationale questions, so other questions more relevant for users could be added.

6.7 Synthesis

This chapter describes both the implementation and evaluation of the conceptual contributions presented in this thesis.

The chapter starts describing UsiExplain, a generic architecture for implementing model-based self-explanatory user interfaces. This architecture relies on five different modules in which the functional core is subdivided, the *Answers Renderer*, the *Interpreter*, the *Processor*, the *Questions Generator*, and the *Questions Renderer*.

The chapter continues with a discussion of the implementation of the architecture. This implementation is based on UsiComp, an integrated framework for the generation of user interfaces at runtime. We have developed this framework in our research group as a basis for different research applications. UsiExplain is one of them.

After presenting UsiComp, the chapter describes UsiCars, a running prototype entirely based on UsiComp. UsiCars is a self-explanatory user interface based on a cars shopping

website. The prototype shows that the unification of different types of questions as described by our approach is possible and feasible.

The chapter then describes an evaluation experiment that has been carried out to evaluate if our model-driven help system is valuable. The experiment makes use of the previous prototype UsiCars. The experiment that we conducted shows that most of the users identifies model-based explanations as potentially useful. The chapter has identified key aspects for further research as, for instance, new possible questions that a help system must include, the weakness of our current implementation, as well as new ways for improving them. The study has also collected some interesting suggestions about usability improvements for help systems, also discussed in the chapter.

The next chapter concludes the research, describing some future work based on all the accomplished research and the obtained results.

7

Conclusions and Future Directions

> *A story has no beginning or end: arbitrarily one chooses that moment of experience from which to look back or from which to look ahead.*

Graham Greene, The End of the Affair,

RELATED PUBLICATIONS

1. DITTMAR, A., GARCÍA FREY, A., AND DUPUY-CHESSA, S. What can model-based ui design offer to end-user software engineering? In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems* (New York, NY, USA, 2012), EICS '12, ACM, pp. 189–194

The chapter starts with the presentation of the major contributions of this thesis, reviewing the research questions and the answers that this work provides for each of them.

The chapter discusses then the advantages and limitations of our approach.

The chapter ends with a number of future directions that have emerged from the work presented in this research, divided into short term perspectives and long term perspectives.

7.1 Summary of the Contributions

This dissertation started with a thorough examination of the literature on help systems in Chapter 2, identifying a lack of systems which can currently accommodate the need of supporting users by covering multiple types of questions at the same time, with a significant reduction of cost.

The dissertation has shown in Chapters 4 to 6 that the model-based solution proposed is possible, feasible, and it has shown to be helpful for most of the proposed questions in the experimentation that we have conducted with real users in a running prototype. This research then proves that design models can be successfully exploited for explanation purposes at runtime.

To come up with this major outcome, a number of contributions have been done:

- The definition of the QAP problem space for comparing different model-based solutions for explanation purposes.
- Design principles for the creation of model-based self-explanatory user interfaces.
- A quality meta-model as a contribution to the UsiXML language.
- Explanation strategies for computing different types of explanations. This thesis proposes six different explanation strategies for six different question types.
- UsiExplain, a conceptual architecture for the implementation of model-based self-explanatory user interfaces.
- UsiCars, an implementation of the conceptual architecture showing the feasibility of the approach.
- A running prototype based on the implementation of the theoretical architecture.
- Investigation of caveats and limitations of the provided explanations (usability issues and explanations acceptance) through an experimentation with real users on the previous prototype.

All of them permit to answer the research questions that have lead our research through this thesis. These research questions and their answers are discussed next.

7.2 Answers to Research Questions

A number of research questions were identified from the problems discussed in chapter 1. Our research provide the following answers to these questions:

Is it possible to generate explanations "for free"? The lack of good help support in most today's software is due to a problem of cost. This thesis has shown through the conceptual contributions described in chapters 4 and 5, that have been put into practice in chapter 6, that the design models can be used not only for building the UI but also as a solution for the generation of support with a minimum cost.

What to explain? Chapter 2 have identified the most common questions that other related works have pointed out as useful. In our research, we have shown that we can successfully answer questions from six different types. The explanation capabilities of a Model Driven UI are restricted by the information available in the models from which the UI is generated, but this research has proven that these models are useful for supportive purposes.

How to explain? Our research provides a generic architecture for answering user's questions. The answers are computed directly from information coming from one or more models, according to an explanation strategy designed specifically for each type of question.

How to present the explanation? Our model-based solution proposes a set of grammars for composing answers, so the user can get explanations in a pseudo-natural language.

Is the provided support valuable? We have conducted an experiment to explore whether the computed support presented to users is valuable, and for most of the question types supported in our prototype, the results shown that they are. However, a number of improvements have been also identified in many areas.

The model-based solution proposed in this research have also a number of interesting properties that are discussed in the next section.

7.3 Advantages of the approach

Our proposed solution provides several properties for those model-based help systems that are developed following our approach. This section discusses these properties and then explores how these model-based help systems compares to other related work according to the QAP problem space defined in chapter 2.

7.3.1 Properties of the Approach

Our approach does not only provide a method for the *Unification* of different question types in the same help system, but it also provides these help systems with the properties of *Introspection*, *Flexibility*, *Distributability*, *Reusability*, and *Customization*. The design principles are also applicable to different architectures and frameworks as a consequences of being an *Open Approach*. These properties are described in the following.

7.3.1.1 Unification of question types

The proposed approach aims to be *universal* in the sense that it unifies different questions types that have already been covered by other previous works, under a single approach in a single architecture. Moreover, the presented approach is not restricted to a specific set of questions but, instead, it is open to new types of questions that designers may consider to include. In the same manner, the presented approach does not set any restrictions or limitations about how to compute the answers, or what sources (models) need to be used to compute answers, meaning that new forms of computing answers can be exploited, added, mixed, all of them under the same principles.

7.3.1.2 Introspection

An introspective help system is able to provide support not only from the models coming from the application but also from its own models, for instance, to answer users' questions about how to use the help system. This is possible because both UIs (application UI and

help UI) are unified by construction as their models conform to the same meta-models, so the same mechanisms for extracting explanations (the Explanation Strategies presented later in our case) can be applied. This means that the same set of questions can be used on the self-explanatory user interface as well with no extra cost.

7.3.1.3 Flexibility for Weaving

The method provides different forms of flexibility regarding how the help UI is integrated into the target application. Help systems can be then classified into three different types regarding how the UIs are mixed:

Weaved - where the help UI and the application UI share the same space of interaction

Non-Weaved - if the help UI runs in a different interaction space

Mixed - where some of the options of the help UI are directly weaved into the application UI, and some others are not.

In the case of Mixed UIs, non-weaved options can be directly accessible from the weaved ones if needed.

This property ensures full flexibility for designers with regard to the *Presentation* of the *Questions* and *Answers* according to the QAP problem space. Our help systems will support both *Intrinsic* and *Extrinsic* presentations by construction for both *Questions* and *Answers*.

7.3.1.4 Distributability

Distributability is the property that allows a UI to be distributed among several devices. Distributing non-weaved UIs is specially easy because the models of the help UI are clearly separated from those of the application UI. This form of flexibility is specially useful for ubiquitous systems where not all the platforms are always available, or we want to require support without stopping other interaction processes. For instance, when some users are playing a film on a laptop, one of them may want to ask about some options of the video player interface without stopping the film. The UI of the help system can be distributed to the smartphone of

this user for this purpose, without stopping the movie and, in consequence, without affecting the experience of the rest of the users.

7.3.1.5 Reusability

Once designers know how to exploit a specific model for supporting purposes, they can easily apply the same procedure to the same kind of models of different applications. For instance, the use case two can benefit of the Where questions of the use case one, as the models of both use cases conform to the same meta-models. Designers can create their model-based help systems once and reuse them everywhere.

7.3.1.6 Customization

The design principle related to how to weave the UI of the help system with the UI of the application allows to perform different customizations of the generated help UIs to fit specific application requirements. For instance, the look and feel of the application UI is normally fixed at the CUI level, using some mechanism based on *stylesheets* or skins stored in the CUI model. Designers would like to preserve the same look and feel for their help systems and applications. This can be accomplished by applying the same mechanism to the CUI model of the help UI. If the help UI is weaved before the CUI level, the look and feel is automatically preserved as there is no specific CUI model for the help system, this is, there is only one CUI model containing both UIs.

7.3.1.7 Open Approach

The design principles presented in this section do not set any restrictions on how designers let end users ask for support, i.e., what is the interaction technique that the user needs to employ to request information. No assumption is made about how the information supporting the end user is provided. There is no restriction on what models designers can use and how they can be exploited. For instance, the computation of the help in the functional core of the help system can be done with rule-based systems based on the application models, or on machine-

learning algorithms.

In the context of our research, we propose the use of Explanation Strategies to provide support to the users through the Cameleon Reference Framework models, employing a subset of the models of the UsiXML language. The architecture earlier presented in this thesis are, however, applicable to any other set of models and frameworks so designers have always the freedom of choice.

7.3.2 Proposed Solution on the QAP Problem Space

Considering the design principles and the properties of self-explanatory user interfaces, we can compare our proposed Self-Explanatory UIs with the existent help systems of the literature that have been reviewed in previous chapters. Figure 7.1 shows how Self-Explanatory UIs are mapped into the QAP problem space presented at the end of the state of the art in chapter two.

As we can see in the image, self-explanatory user interfaces improve the covered previous work in two main areas.

The first one concerns questions about the design rationale, that were not directly covered by any of the related work. With our approach, the design rationale becomes inspectable at design time as well as other information from other models do. This fact not only opens the range of questions by covering those that can be extracted from the underlying design rationale, but it opens new way of supporting end-user programming by directly providing to the users with the design choices that were made at design time.

The second area of improvement is the one related to the structure of the user interface. In fact, as the structure of the user interface is defined in the models from which the source code is derived, questions about the structure of the UI such as *Where an option is* can be easily answered by inspecting these underlying models.

As we can also see in figure 7.1, there is one point that is not covered neither by the previous work on model-based help systems nor by our current propositions. This area belongs to *Initiative* axis. In fact, this is normal because our proposed solution does not make any

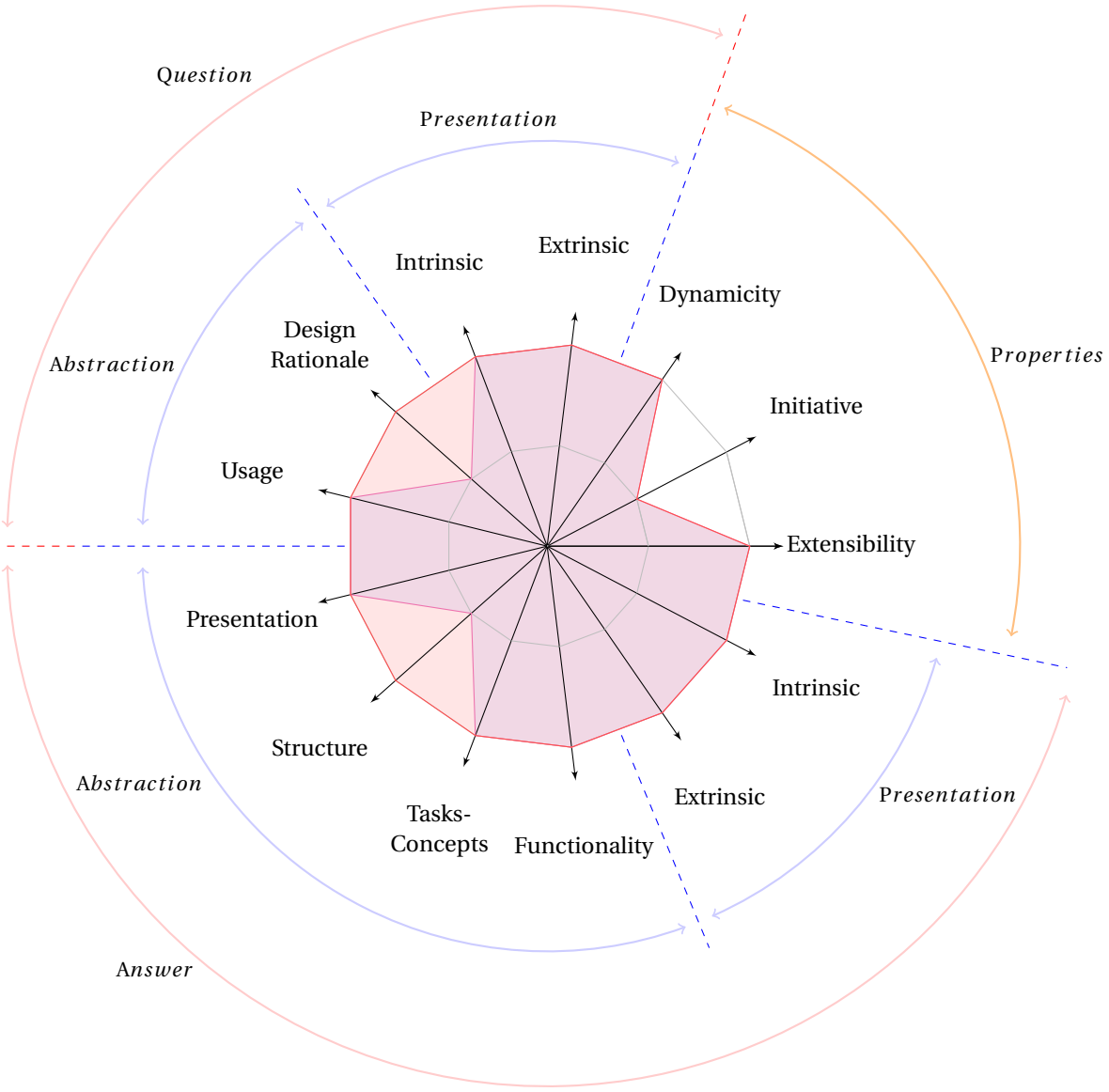


Figure 7.1: Overlapping related work with self-explanatory user interfaces.

assumption about how the interaction technique between the user and the help system is. As we have seen in the previous sections, our approach lets the possibility of choosing the best compromise for the *Initiative* axis to the designers, without forcing a particular solution or interaction technique either for asking a question or for providing the answer.

In the same manner, following our design principles designers can choose the best presentation axis for their applications. In other words, they have now the possibility of presenting *Questions* to users in an *Intrinsic* or *Extrinsic* way, or even combining both of them in the same application if it is necessary. The same applies to the *Presentation* of the *Answers*.

7.4 Limitations of the Approach

This section briefly discusses the limitations that we have found in our approach and the current implementation. The section discusses semantic information, scalability, and usability improvements.

7.4.1 Usability improvements

There is a number of improvements that have been extracted from the experiment that should be considered in further research implementations. First, as some users have pointed out during the experimentation, there is a need of filtering relevant and irrelevant questions that are proposed by the system. The problem here is that one can't never be sure about what questions are relevant for one user, and what questions are not. Assuming an average user with an average amount of knowledge could be a starting point for filtering out those questions that are perceived as irrelevant for most of the users. But the problem is deeper and the user's profile, including mental models of user's knowledge and understanding of the user interface need to be taken into account before adapting the questions proposed by the self-explanatory facility, or adapting the answers that the system provides.

7.4.2 Semantic Information

The models used in our current solution do not include semantic information that could be exploitable for describing concepts, providing definitions, or completing mental concepts of the users. More research is needed for enriching the explanations with semantic information, either for providing new types of questions or enriching those already presented in this research.

7.4.3 Scalability

We did not evaluate how the proposed solution performs in large scale applications with a high number of models. As the list of questions that the self-explanatory facility is able to answer, as well as the answers that it is able to provide, rely all of them on the underlying models using some parts of such models in the computation of the explanation strategies, a high number of models could have a significant impact in the performance of the help system. This potential problem is related not only to the self-explanation solution proposed in this thesis but to model-driven approaches in general.

The scalability problem is not only related to the number of models but also to the complexity of them. For instance, the instance of CUI model containing thousands of objects could perform sensibly worse than an instance with less than a hundred objects.

All these aspects should be empirically evaluated for commercial versions of the proposal solution.

7.5 Future Work

The work presented in this research has revealed a number of potential directions for future work. This section covers some areas of research that are particularly interesting. These areas are: how to improve the usability of help systems, support new question types, support additional sources of knowledge, add initiative to model-based help systems, investigate interaction techniques for requesting/providing information, how to close the loop between

users and designers, implications of supporting design rationale at runtime for learning and end-user programming purposes, and semi-automatic quality guided design for UIs based on the quality meta-model.

This section will briefly discuss each of these directions subdividing them into short term and long term perspectives.

7.6 Short Term Perspectives

This section discusses future work that could obtain results in the short term.

7.6.1 Usability Improvements

Further research needs to be done to explore what is the best presentation and integration of the proposed questions and their related answers. Improving the usability of the help system will lead to a better use of the help system and, in consequence, a better experience with the target application. This research will probably involve techniques for filtering questions out with regard to the user's profile, user's actions, or user's experience.

The presentation of the answers should also be investigated, integrating techniques for exploting answers in different ways. For instance, answers about localisation could take benefit of the model structure to directly propose the desired element to the user instead of providing the path that the user needs to follow to locate such element.

The adaptation of the answer to each user should consider as well the use of different vocabulary if necessary, reviewing the quantity and nature of information provided to each particular user (more information for novice users, less for experts).

7.6.2 Interaction Techniques

The proposed architecture does not set any restrictions about what is the best interaction technique for requesting for information, or what is the best way of providing the explanation back to the user. This could be considered in a global an homogeneous sense where

all the questions and answers are asked and explained in the same way as in the case of our prototype with a dialog containing all the types of questions, or with specific interaction techniques for each type of question or answer. For instance, tooltips have been classically used for answering *What is it for* questions. Model-based approaches should take benefit of these techniques.

7.6.3 Closing the Loop

Tracking what questions are asked by the users of a user interface can help to improve the user interface itself. For instance, in the experiment presented in the previous chapter, almost 120 questions were related to navigational issues. This means that users did not find the option they were looking for easily. The user interface designers could study what question types are asked and at what precise moment, so they can later improve the user interface based on this information.

7.7 Long Term Perspectives

This section discusses future work requiring a more deep research. The discussion focuses on the Initiative Axis unsupported in the QAP problem space, different interaction techniques for inspecting the questions and answers that the self-explanatory system is able to provide, the application of the design rationale related questions as a learning tool for new designers and as a support for end-user programming, and finally, the role of the quality meta-model for automatically generating high quality user interfaces based on quality criteria.

7.7.1 Initiative Axis

Also revealed by the experimentation was the contrast between the good acceptance of *How* questions and the low number of verbatims of such type of questions. This suggests that users find the information useful, because the high acceptance, but they are not thinking of asking for this specific type of explanation most of the time. To overcome this situation, one solution

could be to improve the help UI so it could encourage/propose questions to the user in these situations. This will cover the *Initiative* axis of the QAP problem space.

7.7.2 Quality guided development and evaluation

An interesting research based on QUIMERA, our quality meta-model, relies on the ability of such meta-model to launch transformations that directly modify the element for which the quality is being measured, and, in consequence, directly affecting the quality of the whole system under study. In fact, as these transformations can directly modify parts of or the whole system under study, (for instance, a transformation that chooses between a calendar widget or a TextField for date input) different versions of the same system under study that are issued from different transformations, will present different *Achieved Quality*.

According to this, quality can be semi-automatically re-evaluated with regard to the assessment methods that evaluate different quality aspects of the product. In consequence, different iterations of quality measurements can be semi-automatically done only by applying different transformations each time, and evaluating the obtained quality for each transformation.

This open new ways to explore quality guided design processes in which quality becomes an active factor that semi-automatically guides the design of the system under study according to the quality requirements specified by the quality expert through quality criteria.

7.7.3 Supporting New Question Types

Previous chapter has revealed a list of new types of questions that are currently unsupported by our system. A number of them referred to *What if questions* that can provide information about the possible side effects of using an option in the UI. These kind of questions can probably be answered by analysing the operators of the task model and how they are transformed to CUI elements, i.e., what elements of the CUI model become active/inactive as we enable/disable new tasks.

As revealed in the experiment, a number of questions about confirmation procedures

were also identified . Supporting questions about confirming and validating the user actions can help to overcome those situations were the feedback provided by the user interface is not enough and the user requires validation or confirmation from the application.

Another type of questions, also identified in the experiment, concerns semantic definitions, that ask for the meaning of a concept that appears in the user interface but the user does not understand. For instance, in the experiment, there were a number of specific car-related terminology that were difficult to understand for non expert users.

A last interesting type of question, already exploited in recommender systems, is that explaining the differences between some entities or concepts of the UI, for instance, *What is the difference between the pack sport and the pack excellis?* Supporting this type of answer could help to improve the user's understanding of the UI, and thus, he/she confidence on the application.

7.7.4 Supporting New Sources of Knowledge

In this research we have explored how different models can contribute to support users at runtime. These models are the four models representing the different levels of abstraction of the Cameleon Reference Frawework, plus some other models that we have identified as relevant for this purpose such as the mapping model, the quality model, or the QOC model.

Other models could be considered as an alternative source of knowledge. For instance, we have already discussed in the section findings of the qualitative evaluation that ECA models [153] or Command Object Models [100] could positively improve the answers for behavioural questions such as *Why does it happen?* and *Why it does not happen?*.

New types of questions could take benefit of new models including semantic information, for instance those questions related to definitions of concepts. This could probably be accomplished either by adding new models containing this type of knowledge or by connecting the UI with sources that already supply the necessary semantic information, such as the Internet. For instance, using queries to semantic browsers like Wikipedia[1] or the Wolfram Al-

[1] `http://www.wikipedia.org/`

pha engine[2] employed by the Personal Assistant *Siri*. This opens a new research about how to improve model-based explanations with non model-based sources that already works for a specific kind of question. Is this way worth exploring? Is there a model or set of models that can supply the same information? What models? What explanation strategies do we need to define to correctly exploit such models?

7.7.5 Design Rationale for Learning / End-User Programming

With the integration of design rationale questions, users can better understand the underlying reasons of the design decisions made by the designers of the user interface. This open a new research area for learning or training new designers so that they can access in real time to the rationale of the UI, but also for end-user programming, explaining to the users why the UI is the way it is, so they can better understand what to modify, how to do it, and the implications that such modification involve. We have already started to explore the implications of model-based approaches for end-user programming in [30], where we discuss some interesting annotations on the core models that help to decide about the design space for end users and some Extra-UI[3] design patterns to support appropriate representations of this design space. From a model-based approach, the questions and answers presented by a self-explanatory UI could be considered as an extra-UI because they provide a different representation of the underlying models. End-user programming will help to explore other different representations, eventually providing access to the full models of the UI if the user is an expert, or not only providing explanations about the UI but directly helping users to manipulate the models with an appropriate extra-UI with self-explanation support.

[2] `http://www.wolframalpha.com/`
[3] An Extra-UI is a UI that represents and provides control over a UI [138].

Appendices

Appendix A

Specification of the Quimera Quality Meta-Model

This appendix describes the meaning of the classes of the quality meta-model.

QualityModel The QualityModel meta-class defines the representation of a Quality Model. Its attributes are:

- name (String): Specifies the name of the Quality Model.
- standard (Boolean): Specifies whether the current instance of the Quality Meta-Model represents a quality standard or not. If true, the quality model represents a standard such as ISO 9241-110. This means that the model is composed only of instances of QualityModel, Criteria, Attribute and CriterionAssociation meta-classes.

Criterion The Criterion meta-class describes how the Quality Meta-Model is composed. A quality model is composed of criteria, that can be recursively decomposed into sub-criteria as well through the CriterionAssociation class. This representation allows to instantiate different standards from different communities such as the Software Engineering community (for instance to evaluate the quality of the source code) or the HCI community (for example, instantiating the four layers of QUIM[17].)

Its attributes are:

- *name* (String): Defines the name of the Criterion. Example: Usability for the task.
- *problem* (String): Defines the problem the Criterion is dealing with.
- *context* (String): Specifies the context in which the Criterion applies.

Attribute Defines a characteristic of a Criterion. Its attributes are:

- *name* (String): It allows to specify one or more attributes for a Criterion.
- *cardinality* (Unsigned Int): Defines the cardinality of the attribute. By default, the cardinality is one.
- *type* (String): Defines the type of the attribute.
- *value* (String): Holds the value of the attribute.

CriterionAssociation The CriterionAssociation is an abstract element that defines the relationship of the Criterion accordingly to the definition of the Quality Model. Its attributes are:

- type (AssociationType): Defines the type of the association. This allow to define how different Criteria are related. Possible values: SupportedBy, UnsupportedBy, DiscriminatedBy. A Criterion can support other criteria (for instance, in QUIM a factor at the Factor level is supported by criteria from the Criteria level). It can be discriminated by other Criterion, typically when two criteria are in conflict, or the relationship can be unsupported when two Criteria are not in conflict but there is no support between them.

Recommendation A Recommendation is a positive assessment that corresponds to one or more criteria. For instance, the Recommendation says that good quality can be achieved by maximizing the number of criteria that are satisfied by a given UI. Figure 5 shows how different Metrics are used for the same Recommendation. A Recommendation can be decomposed or rewritten in sub-recommendations through the isRewrittenBy association. Its attributes are:

- *name* (String): Defines the name of the Recommendation.
- *description* (String): Explains the Recommendation.
- *author* (String): Defines the name of the author of the Recommendation, to keep trace of the different Recommendations each quality expert has done.
- *weight* (Integer): Defines the current weight of a Recommendation. The weight allows the quality expert to model how important a Recommendation is with regard to others.
- *weightDescription* (String): Explains how the weight is interpreted.

RecommendationAssessmentMethod This class represents the way in which the quality expert or the system itself can determine if a Recommendation is accomplished or not. A RecommendationAssessmentMethod is specialized in Metrics or Practices. It can be subjective or objective. Its attributes are:

- *name* (String): Defines the name of the Metric or Practice to be used.
- *description* (String): Explains the Metric or Practice, describing the formula and its different elements in the case of a Metric, or what does the Practice involve and how to know if it has been followed or not.
- subjective (Boolean): Explains whether the measurement is subjective (true) or objective (false). Note that even subjective evaluations can be measured quantitatively (for instance by Metrics) or qualitatively (for instance by a Practice). The attribute subjective makes explicit this distinction and allows quality experts to cover both dimensions as depicted in the figure A.1.

Metric Express how to compute a numerical value for a given Artifact. Metrics are associated to NumericalResults. Its attributes are:

- *author* (String): The author of the metric.
- *numericalExpression* (String): Defines the associated formula for the metric.

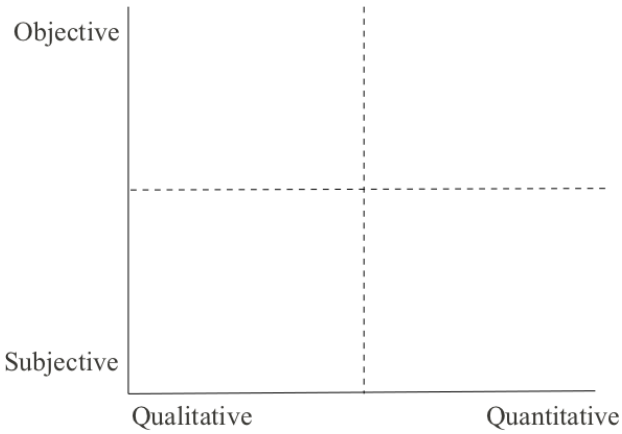


Figure A.1: Quality and Subjectivity.

Limits Holds the desired values for a given metric. Attributes:

- *lower* (Double): Defines the minimum value the metric is desired to achieve.
- *upper* (Double): Defines the maximum value the metric is desired to achieve.
- *interpretation* (String): Explains how to interpret ate the limit values.

Practice The Practice meta-class represents Practices, i.e., proven processes or techniques that organizations or persons have found to be productive and useful to ensure a good level of quality (Good Practices), or unproductive and unusable (Bad Practices). "Design patterns" are an example of the first one, whilst "Spaguetti code" is an example of the second one.

Its attributes are:

- *practiceType* (PracticeType): Defines if the Practice is applicable to a Process or a Product. Possible values: process, product.
- *patternType* (PatternType): Defines if the Practice represents a Pattern or an Antipattern. Possible values: pattern, antipattern.

LocalResult Holds the result of an AssessmentMethod. Its attributes are:

- *value* (Float): In the case of Metrics, the value represents the result of the computation of the numericalExpression of the Metric. In the case of a Practice, the value attribute represents the percentage in which a Practice is satisfied. AssessmentMethod: This meta-class specifies how to compute Metrics and Practices together. The global quality of a SUS is computed through AssessmentMethods. Its attributes are:
- *name* (String): Defines the AssessmentMethod name.
- *formula* (Metric U Practice): Defines how the different Metrics and Practices are combined to computed the result.

GlobalResult This meta-class holds the global quality of a given SUS. The result is computed using the Results obtained from Metrics and Practices according to the specific AssessmentMethod. Its attributes are:

- *interpretation* (String): Express how the result of the AssessmentMethod must be interpreted.
- *result* (Float): Holds the global quality value of a SUS according to an AssessmentMethod.
- *timestamp* (Date): Information regarding when the quality result has been computed.
- *version* (Float): Current version of the SUS on which the quality value has been computed.

Transformation The Transformation meta-class refers to a TransformationUnit from the Transformation Meta-Model. This TransformationUnit will manage all the necessary TransformationUnits (if more than one is required) and it will establish the order in which they must be triggered accordingly to the Transformation Meta-Model. Please, refer to the Transformation Meta-Model section for more information about Transformation Units.

Artifact The Artifact meta-class refers to any element of the Software Development Life Cycle, such as code, classes of a model or the model itself. In this case, it is represented by the Meta- ModelElement from the Transformation Meta-Model. Please, refer to the Transformation Meta- Model section for more information about Meta-ModelElements.

ContextModel As a same Quality Criterion can have different quality interpretations regarding the context in which the interaction is taking place, the Quality Meta-Model needs to know exactly what the context is and how it is defined. Linking the Context Model to the Recommendation meta-class will allow to the quality experts to define different Recommendations regarding the different contexts in which the interaction can occur.

Appendix B

Meta-Models

This annex describes the different meta-models on which the implementation is based. All the meta-models have been implemented in ecore, and thus, ecore representations are used for presenting them. The covered meta-models are those originated from the Cameleon Reference Framework, i.e., the task meta-model, the AUI meta-model, and the CUI meta-model. A domain meta-model, also described, is added for representing the concepts manipulated through the tasks. The link between tasks and concepts, as well as the transformations of each element from one model to another, is kept in a model conforming to the mapping meta-model, which is also used in QOC already presented in chapter 4.

B.1 Tasks

Figure B.1 shows the task meta-model in ecore notation. A task model is composed of tasks. The task model is a tree-form model in which a task can be related to other tasks from the same level -or sister tasks- by binary operators. A task can also have children tasks through the *CompositionRelationship* meta-class. A task can also have an unary operator to indicate for instance if the task is optional or iterative.

As shown in the figure, the task meta-model has been designed to produce CTT based task models.

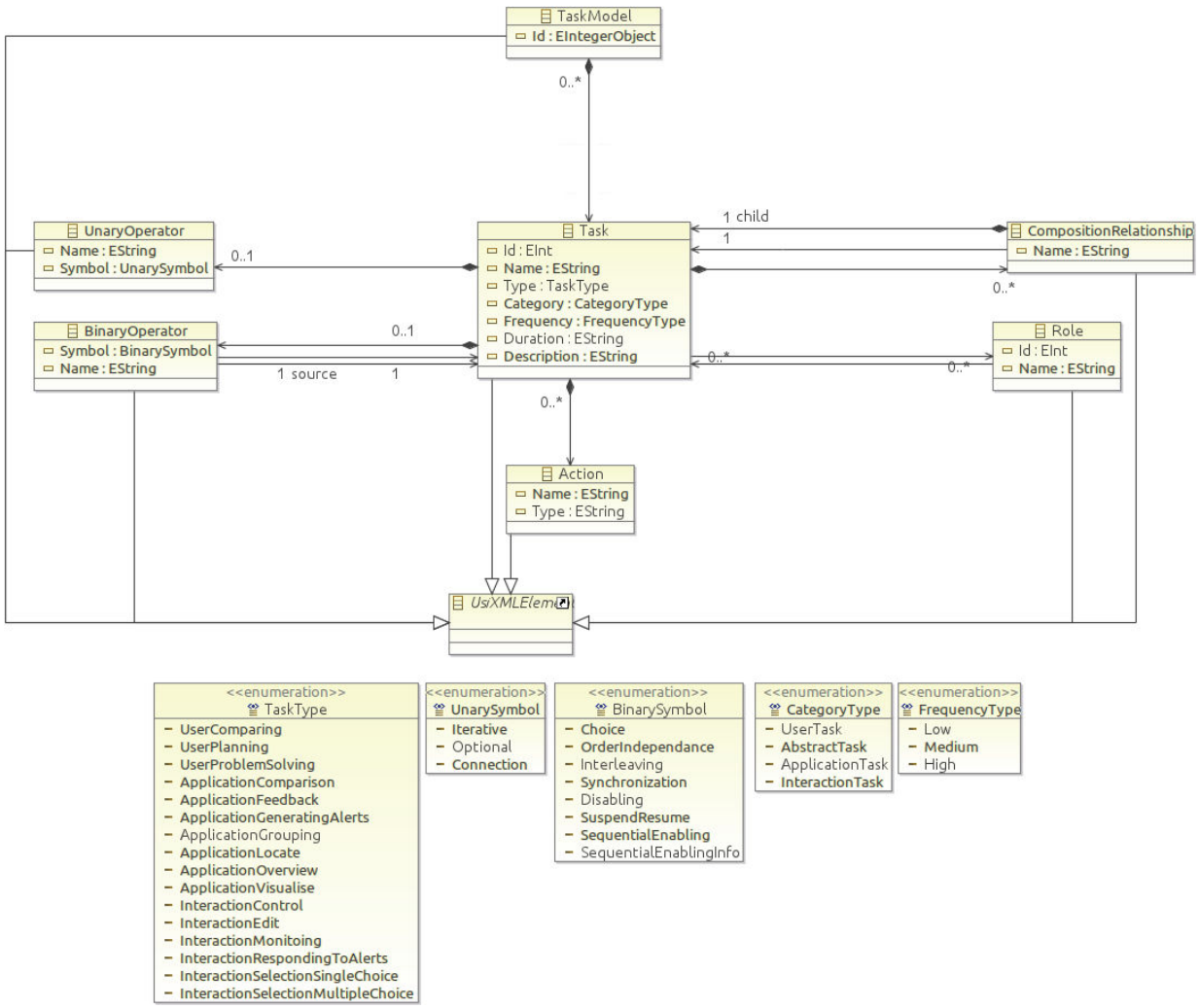


Figure B.1: Task meta-model implementation in Ecore.

B.2 Domain

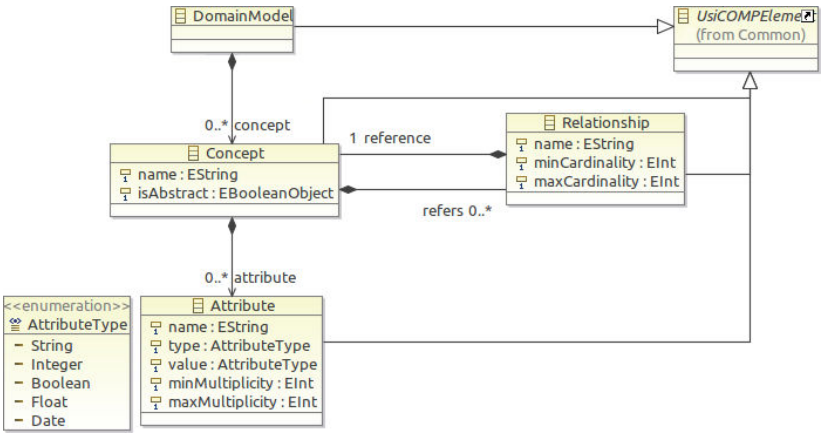


Figure B.2: Domain meta-model implemented in Ecore.

The domain model represents a set of concepts related between them in any arbitrary way (figure B.2). Associations between concepts are represented by the *Relationship* meta-class. Each concept can have attributes if necessary by instantiating the *Attribute* meta-class.

B.3 AUI

An AUI meta-model is composed of *AbstractInteractorUnits* (figure B.3). Each *AbstractInteractionUnit* is defined either as an *AbstractCompoundUI* or as an *AbstractElementaryUI*, following a composite pattern. This produces tree-form aui models. Navigation between different *AbstractInteractionUnits* is done through *AbstractRelationships*.

An *AbstractElementaryUI* can be hierarchically defined either as an *AbstractDataUI* or as a *AbstractSelectionUI*, which is a particular case of *AbstractDataUIs*.

B.4 CUI

Figure B.5 shows our current implementation of the CUI meta-model. A CUI model can be either *Tactile*, *Graphical* or *Vocal*. For our current implementation, multi-modal UIs are out

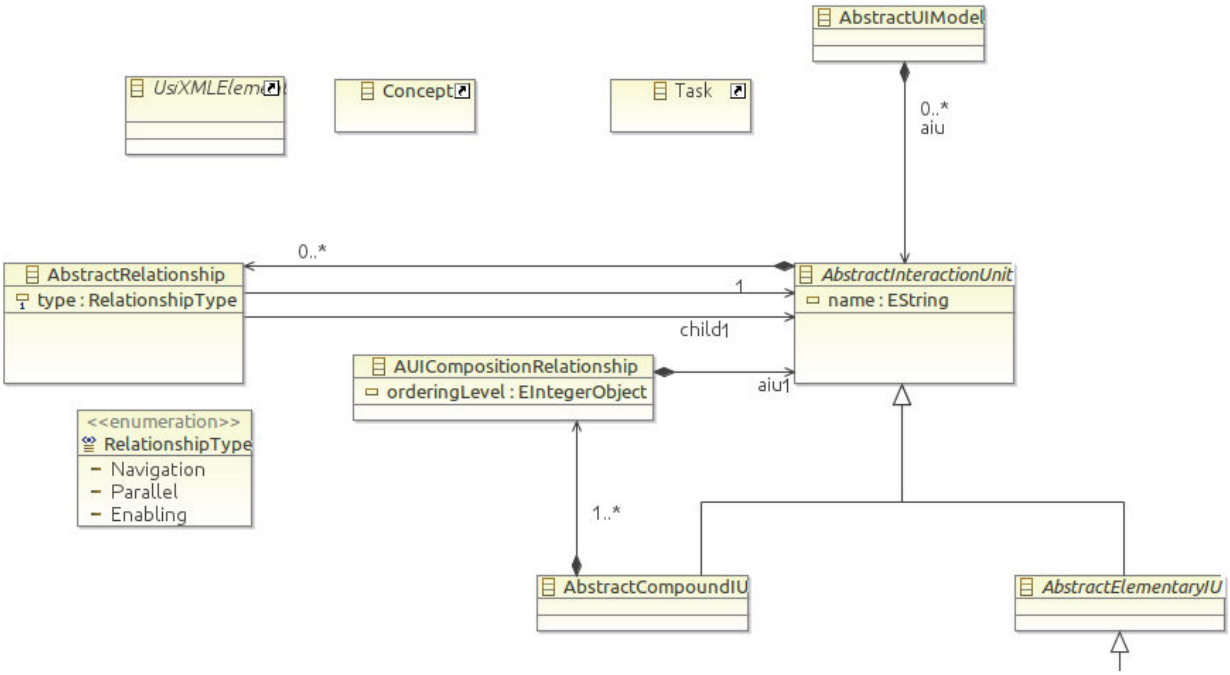


Figure B.3: The implementation of AUI meta-model in Ecore.

of the scope of this research. A *Graphical* UI is composed of windows. Each *Window* can be decomposed into sub-windows if necessary. A *Window* contains different widgets arranged into *Layouts* or *Panels*, and it can also contain a *MenuBar*. We have modeled different standard widgets such as *Buttons*, *ToolBars*, *TextFields*, *ComboBoxes*, *ListBoxes*, *Images*, *Labels* and others. Each of these widgets has been decomposed into elementary elements when possible. For instance, a *Button* showing the message "Accept" does not contain a Caption attribute but, instead, a Label which contains itself the aforementioned text.

B.5 Mapping

As reader could notice in the previous meta-models, all the elements of all the different meta-models inherits from the *UsiXMLElement* meta-class defined in the Mapping meta-model (figure B.6). This meta-class has been defined to provide a tracking mechanism through mappings. Mappings are not always useful for tracking transformations (for instance, what is the

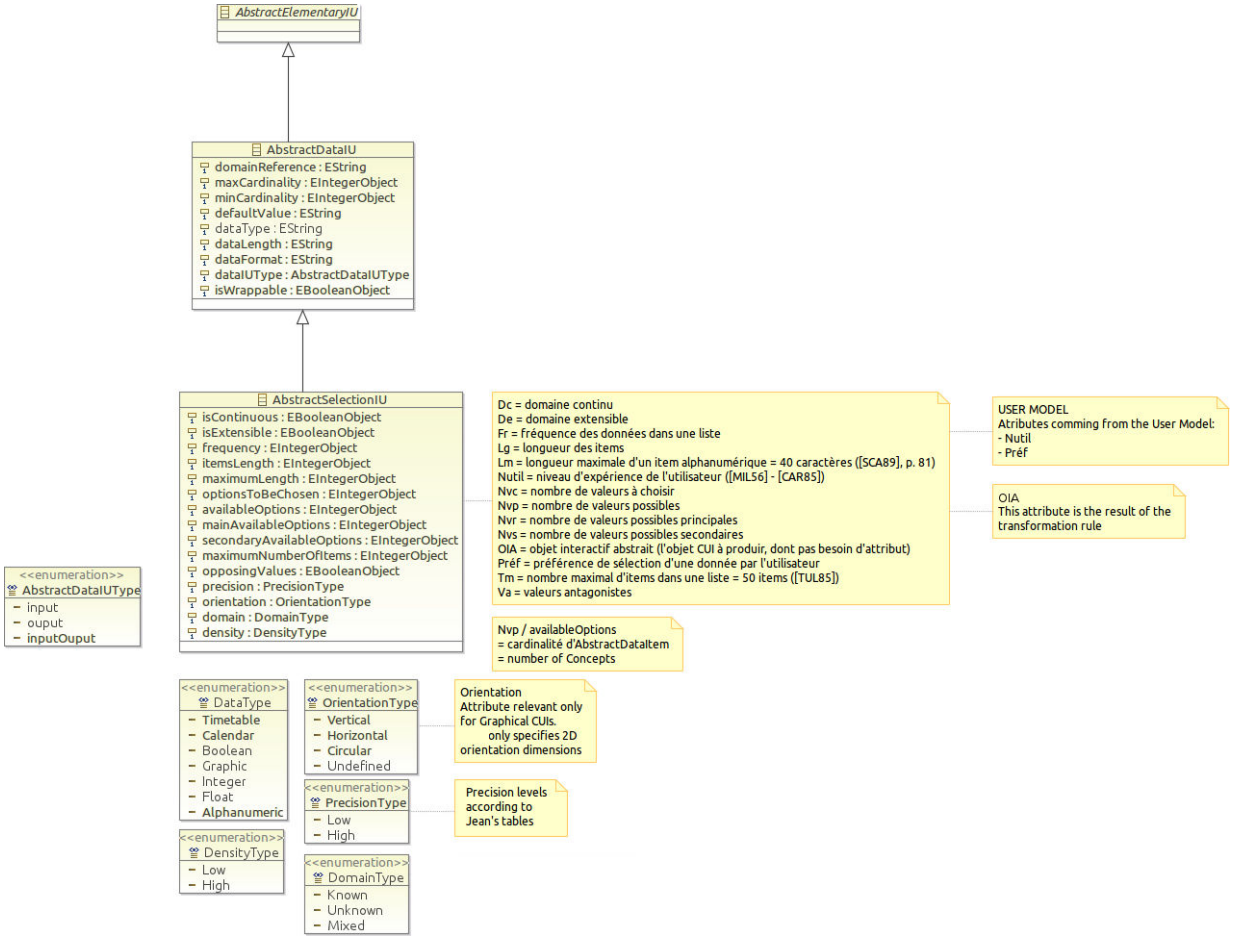


Figure B.4: Detail of the hierarchy decomposition of the AbstractElementaryUI meta-class in the AUI meta-model.

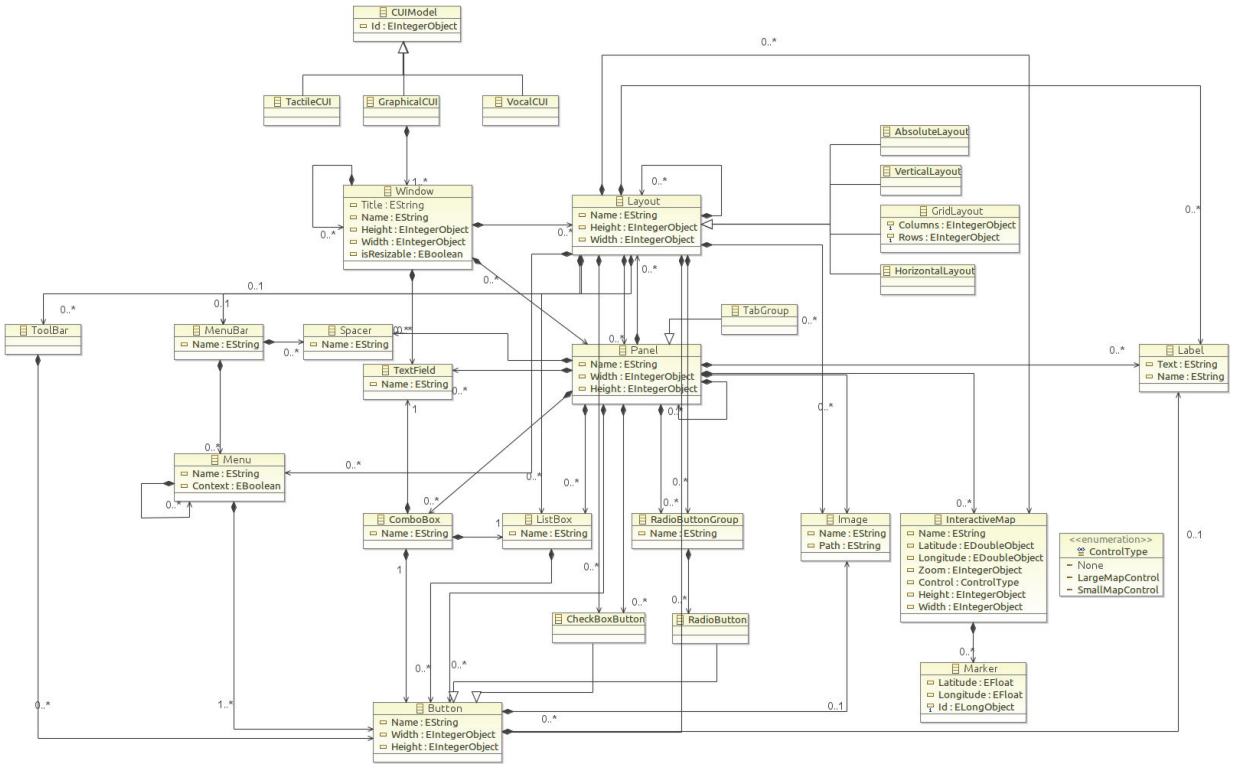


Figure B.5: CUI meta-model implementation in Ecore.

aui element in which a task has been transformed) but also for defining inter-model relationships.

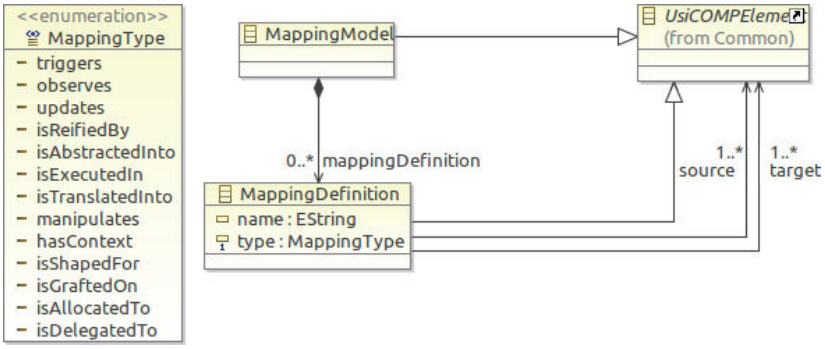


Figure B.6: Mapping meta-model implementation in Ecore.

An example of such situation is the mapping between tasks from the a task model and the manipulated concepts from the domain model, allowing designers to specify what concepts are directly manipulated by each task.

B.6 QOC

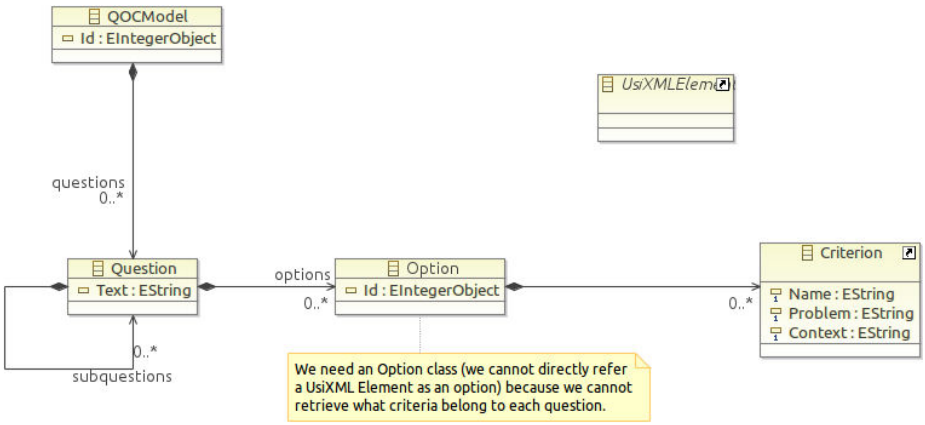


Figure B.7: QOC meta-model implementation in Ecore.

According to the QOC notation, a QOC model is composed of questions about certain design options. We model this options as a mapping to any *UsiXMLElement* as shown in

figure B.7. Criteria is directly linked to the *Criterion* meta-class from the Quality meta-model presented in chapter 5.

Appendix C

Contributory Papers

1. García Frey, A., Calvary, G., Dupuy-Chessa, S., and Mandran, N. Model-based self-explanatory UIs for free, but are they valuable? In *Proceedings of the 14th IFIP TC13 Conference on Human-Computer Interaction (INTERACT'13), 2-6 September 2013, Cape Town, South Africa* (2013), Springer,
2. García Frey, A., Calvary, G., and Dupuy-Chessa, S. Users need your models! exploiting design models for explanations. In *Proceedings of HCI 2012, Human Computer Interaction, People and Computers XXVI, The 26th BCS HCI Group conference (Birmingham, UK)* (2012)
3. García Frey, A., Céret, E., Dupuy-Chessa, S., Calvary, G., and Gabillon, Y. Usicomp: an extensible model-driven composer. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems* (New York, NY, USA, 2012), EICS '12, ACM, pp. 263–268
4. García Frey, A., Céret, E., Dupuy-Chessa, S., and Calvary, G. QUIMERA: a quality metamodel to improve design rationale. In *Proceedings of the third ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2011)* (2011), ACM Press, pp. 265–270
5. García Frey, A., Céret, E., Dupuy-Chessa, S., and Calvary, G. QUIMERA - toward an unifying quality metamodel. In *Congrès INFORSID'11 (Lille, France, May 2011), 6 pages.* (2011)
6. García Frey, A., Calvary, G., and Dupuy-Chessa, S. Self-explanatory user interfaces by

model-driven engineering. In *Proceedings of the CHI'10 Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI'10)* (2010), pp. 1–4

7. GARCÍA FREY, A., CALVARY, G., AND DUPUY-CHESSA, S. Xplain: an editor for building self-explanatory user interfaces by model-driven engineering. In *Proceedings of the second ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2010)* (2010), ACM Press, pp. 41–46
8. GARCÍA FREY, A., CALVARY, G., AND DUPUY-CHESSA, S. Self-explanatory user interfaces by model-driven engineering. In *Proceedings of the second ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2010)* (2010), ACM Press, pp. 341–344
9. DITTMAR, A., GARCÍA FREY, A., AND DUPUY-CHESSA, S. What can model-based ui design offer to end-user software engineering? In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems* (New York, NY, USA, 2012), EICS '12, ACM, pp. 189–194

List of Figures

1.1 A car shopping website. 3

1.2 Thesis structure. 10

2.1 Different roles of agents. Adapted images from [85]. 18

2.2 Clippy. 19

2.3 Evolution of question classifications by authors in the last forty years. After 1990, most of the classifications reuse the same question types. 21

2.4 An example question/answer regarding the airport security wait time at SeaTac Airport (image from [102]. 31

2.5 Siri in action. 32

2.6 Apple location facility. 34

2.7 Avatars providing multimodal information at the Birmingham Airport (picture taken in September 2012) 35

2.8 Authentication dialogue of a broker bank. 38

2.9 Caption 43

2.10 Examples of different types of help according to the Presentation axis. *Left*: Intrinsic help (Clippy from Microsoft Word©). *Right*: Extrinsic help (An online help dialogue presented outside the application that contains all the supportive information ordered by categories). 45

2.11 Hypothetical system under study supporting answers about the Representation and Structure of the system, and without support for tasks-concepts and functionality related answers. 51

2.12 Left: Crystal answers "Why is the 'p' bold?" by highlighting the relevant user interface controls. Right: The "Why?" menu. Images adapted from [100]. 52
2.13 Myers' Crystal help system. 53
2.14 The PervasiveCrystal system ([153]) answering a Why question. 55
2.15 Vermeulen's PervasiveCrystal system. 56
2.16 The Cartoonist system ([141]) in action: "the animated character of the mouse is shown trailing the cursor which is being moved to click on the *Create a NAND gate* command icon." 57
2.17 Sukaviriya's Cartoonist help system. 58
2.18 Image and description from [79]. This explains "Why" the application inferred "Breakfast". The evidences are indicated by the area of bubbles around the corresponding sensors in the floorplan. 59
2.19 Lim's Intelligibility Toolkit. 60
2.20 Overlapping the related work. Identifying potential areas of interest. 63

3.1 Relationship between model approaches followed in this research. 69
3.2 Example of model. The entity *Person* can own an unlimited number of *Cars*. . . . 71
3.3 Example of meta-model. An *Association* is related to *Entities* thorugh *AssociationEnds*. 72
3.4 Model-Driven Architecture. 73
3.5 Four layers pyramid showing each of the OMG levels. 74
3.6 Model transformation. 75
3.7 Example of a model transformation. Generation of java code from UML through an ATL transformation. 76
3.8 PIM model (left) and PSM model (right). 76
3.9 General pattern of the MDA process. 77
3.10 Example of the MDA process. Two PIM source models are transformed together to generate a single PSM model. 78
3.11 The Cameleon Reference Framework ([18]) 81

3.12 Cameleon levels from final user interface to tasks level. Two transformations are drawn with straight lines. The source and the target of the transformations are outlined with circles. 82
3.13 Example of task model. Modelisation of a help system that supports three different types of questions that the user can ask. 83
3.14 Abstract UI model (AUI). 84
3.15 Concrete model (CUI). 84
3.16 Final UI (FUI). 85
3.17 Relevant works on quality in their years of publication. 88
3.18 McCall's quality model: 11 quality factors are decomposed into 23 quality criteria. 89
3.19 Boehm's quality model. 90
3.20 Dromey's implementation quality model. 91
3.21 ISO 9126 quality model for external and internal quality. 93
3.22 ISO 9126 quality model for quality in use (characteristics). 94
3.23 QUIM Structure and Usages. 95
3.24 Example of components relationships. 95
3.25 Finne's quality meta-model extracted from [40]. 96
3.26 Levels of abstraction in quality modelling according to [40]. 97

4.1 A help message leads the user through the UI. English translation: "To select the Non-Smoker Kit use the 'Select Non-Smoker kit' CheckBox. The 'Select Non-Smoker kit' CheckBox is located in the 'Select Extra Equipment' panel." 103
4.2 Norman's Theory of Action. 105
4.3 Design models influence the designer's behaviour in the interaction process. . . 106
4.4 Gulf of Quality. 106
4.5 Explanation through a query paradigm. 108
4.6 Global approach for Self-Explanatory help sytems. 110
4.7 Example of the global approach. 111

4.8 The design principles explain how to build a self-explanatory UI based on models that is able to answer questions about the UI of the Application. 112
4.9 Building a model-based self-explanatory user interface. 114
4.10 Building the UI of an example application according to the Cameleon Reference Framework. Left: Source models. Right: Excerpt of the Final UI. 115
4.11 Generation of a model-based help system. Each column represents the models and transformations that produces the source code for the application UI (left) and the help UI (right). Different combinations for weaving both UIs are represented in the centre of the image. 116
4.12 The car shopping website example. 119
4.13 Models used for generating questions (left) and answers (right). 120
4.14 Excerpt of the task model of the car shopping website. 121
4.15 Explanation strategy for *"How"* questions. The question identified as *How* (1) is used by the explanation strategy to locate the task of the question (2), then to follow the mappings to reach the widgets at CUI level (3), retrieve these widgets (4) and provide the answer back (5). 123
4.16 Example of information retrieving for *How* questions. Mappings are followed from the Task model to the AUI (1) to find the AUI elements in which the task is transformed. Then, CUI elements are retrieved with same procedure (2). 124
4.17 Sequence diagram for computing *How* questions 125
4.18 Excerpt of the CUI model of the car shopping website described in UML (top) and represented using a mockup (bottom). 126
4.19 Explanation strategy for *"What is this for"* questions. The question identified as of type *Purpose* (1) is used by the explanation strategy to locate the CUI element asked in the question (2). Once the CUI element has been located, the explanation strategy follows the mappings to reach the task at the Task level (3), retrieving the task (4) and providing the answer back to the user (5). 128

4.20 Information retrieving for *What is it for* questions. Mappings are followed from CUI to AUI (1) to retrieve the AUI element. Then, from AUI to Tasks (2) to find the task at the source of the transformation chain. 129
4.21 Sequence diagram for computing *What is it for* questions 129
4.22 Excerpt of the CUI model of the car shopping website. 131
4.23 Explanation strategy for *"Where"* questions. As the question is identified as of type *Where* by the help facility (1), it is used by the explanation strategy to locate the CUI element involved in the question (2). Once the CUI element has been located, the explanation strategy retrieves the container CUI element inside the same CUI element (3). Once the container is retrieved (4) its information is used by the help facility to compose the final answer (5). 132
4.24 Excerpt of the CUI of the car shopping website. All the CUI elements such as Tabs and CheckBoxes have one parent container by construction. 133
4.25 Sequence diagram for computing *Where* questions 134
4.26 Configuration options under the Equipment tab. 135
4.27 Available tasks at the state shown in figure 4.26. 136
4.28 Explanation strategy for *"What can I do now"* questions. The question is identified as of type *Availability* by the help facility in (1). The explanation strategy locates the current active task inside the task model (2). The algorithm for finding all the currently available tasks is then applied (3). The list of available tasks is recovered by the explanation strategy in (4). Finally, the help facility composes the final answer (5). 137
4.29 Sequence diagram for computing *What can I do now* questions 138
4.30 Unexpected behaviour (left) and expected result (right) of the Visualization tab. The user gets no feedback about why the unexpected behaviour happens. 139
4.31 Excerpt of the Task model of the car shopping website. The task "Visualize the result" is enabled by all the previous sisters tasks because of the *Sequential Enabling* operator represented by "»". 140

4.32 Explanation strategy for *"Why I can't"* questions. Questions of type *Behavioural* are treated (1) by the explanation strategy by locating the disabled task inside the task model (2). The algorithm for finding the LOTOS operator that activates this task is then applied (3). This information is recovered by the explanation strategy (4) and the help facility uses it to compose the final answer (5). 141

4.33 Sequence diagram for computing *Why I can't* questions. The algorithm looks for sister and mother tasks enabling an unreachable task. 143

5.1 Example of a QOC model for choosing between two types of interactors 147

5.2 Quimera: the quality meta-model 150

5.3 Modelisation of Quality Perspectives 153

5.4 Global quality (green top box), Local quality (pink bottom box), and their relationship to Objects, Methods and Results. 156

5.5 A part of the quality model of ergonomic criteria. 158

5.6 Subset of the objects of Symphony evaluation model. 160

5.7 Example of an instance of the quality meta-model. The quality model is the standard ISO 9241-110. The instance shows a subset of seven criteria from this standard. 162

5.8 Graphical representation of the connection between the quality model (right) and a design rationale notation, QOC in this example (left). 163

5.9 Sequence diagram for computing *Design Rationale* questions 167

5.10 QOC and Quality models linked through quality criteria. Design rationale questions are directly retrieved from the QOC model. Answers are provided according to the criteria that support (Assessment) the selected option. 168

6.1 The Self-Explanatory User Interface consist of the UI of the application (right) plus the self-explanatory facility (middle) being both of them model-based UIs. . 173

6.2 Generic architecture for model-based self-explanatory help systems. The functional core of the help UI accesses any (meta-)model at runtime. 174

6.3 Overview of the OSGi Layers 177
6.4 UsiComp software architecture: meta-models, models and transformations at the heart of both design time (IDE for designers) and runtime (FUIs for end-users). 179
6.5 UsiComp Development Environment. From Top to Bottom: Task model, AUI model, CUI model. Transformations are represented by arrows. 180
6.6 Two examples of the UsiComp extensibility. A task model is generated from an external tool called Compose. A CUI model can also be generated from a mockup instead of transforming the AUI model. 184
6.7 Relationship between the UsiExplain generic architecture and the UsiComp framework. 185
6.8 Screenshot of the prototype. Choosing the model. 188
6.9 Screenshot of the prototype. Selecting the external colour. 189
6.10 Self-Explanatory dialogue showing the full list of types and questions. 190

7.1 Overlapping related work with self-explanatory user interfaces. 208

A.1 Quality and Subjectivity. 221

B.1 Task meta-model implementation in Ecore. 225
B.2 Domain meta-model implemented in Ecore. 226
B.3 The implementation of AUI meta-model in Ecore. 227
B.4 Detail of the hierarchy decomposition of the AbstractElementaryUI meta-class in the AUI meta-model. 228
B.5 CUI meta-model implementation in Ecore. 229
B.6 Mapping meta-model implementation in Ecore. 230
B.7 QOC meta-model implementation in Ecore. 230

References

[1] *SIGMOD Rec. 33*, 1 (2004). 68

[2] ABRAHÃO, S., IBORRA, E., AND VANDERDONCKT, J. Usability evaluation of user interfaces generated with a model-driven architecture tool. In *Maturing Usability*, E.-C. Law, E. Hvannberg, and G. Cockton, Eds., Human-Computer Interaction Series. Springer London, 2008, pp. 3–32. 145, 148

[3] ACT-NET CONSORTIUM, C. The active database management system manifesto: a rulebase of adbms features. *SIGMOD Rec. 25*, 3 (Sept. 1996), 40–49. 31

[4] AKOUMIANAKIS, D., SAVIDIS, A., AND STEPHANIDIS, C. Encapsulating intelligent interactive behaviour in unified user interface artefacts. *Interacting with Computers 12*, 4 (2000), 383–408. 2, 19

[5] ALEXIS CÖTÉ M. ING, M., PHD, W. S., AND GEORGIADOU, E. Software quality model requirements for software quality engineering. In *Software Quality Management and INSPIRE Conference (BSI) 2006* (2006). 89, 90, 91

[6] ANDRENUCCI, A., AND SNEIDERS, E. Automated question answering: review of the main approaches. In *in the Third International Conference on Information Technology and Applications* (2005), pp. 4–7. 21

[7] BASTIEN, C., AND SCAPIN, D. Ergonomic Criteria for the Evaluation of Human-Computer Interfaces, 1993. 98, 154, 157

[8] BELLOTTI, V., AND EDWARDS, K. Intelligibility and accountability: human considerations in context-aware systems. *Hum.-Comput. Interact. 16*, 2 (Dec. 2001), 193–212. 59

[9] BENYON, D., AND MURRAY, D. Adaptive systems: from intelligent tutoring to autonomous agents. *Knowledge-Based Systems 6*, 4 (Dec. 1993), 197–219. 17

[10] BERLAGE, T. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Trans. Comput.-Hum. Interact. 1*, 3 (Sept. 1994), 269–294. 52

[11] BÉZIVIN, J. On the unification power of models. *Software and System Modeling 4*, 2 (2005), 171–188. 70, 77

[12] BOEHM, B. W., BROWN, J. R., KASPAR, H., LIPOW, M., MCLEOD, G., AND MERRITT, M. *Characteristics of Software Quality*. TRW series of software technology. TRW Systems and Energy, Inc. (1973); also published by North-Holland, Amsterdam., 1978. 88, 149, 151

[13] BOOCH, G. *Object oriented design with applications*. Benjamin/Cummings series in Ada and software engineering. Benjamin/Cummings Pub. Co., 1991. 68

[14] BOWEN, J., AND REEVES, S. Modelling user manuals of modal medical devices and learning from the experience. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems* (New York, NY, USA, 2012), EICS '12, ACM, pp. 121–130. 2

[15] BRADSHAW, J. *Software agents*. AAAI Press Series. AAAI Press, 1997. 18

[16] BUCHANAN, B., AND SHORTLIFFE, E. *Rule-based expert systems: the MYCIN experiments of the Stanford Heuristic Programming Project*. Addison-Wesley series in artificial intelligence. Addison-Wesley, 1984. 15

[17] CALVARY, G., COUTAZ, J., THEVENIN, D., LIMBOURG, Q., BOUILLON, L., AND VANDERDONCKT, J. A unifying reference framework for multi-target user interfaces. *INTERACTING WITH COMPUTERS 15* (2003), 289–308. 48

[18] CALVARY, G., COUTAZ, J., THEVENIN, D., LIMBOURG, Q., BOUILLON, L., AND VANDERDONCKT, J. A unifying reference framework for multi-target user interfaces. *Interacting With Computers Vol. 15/3* (2003), 289–308. 80, 81, 235

[19] CARLIER, A. *Management de la qualité pour la maîtrise du SI: ITIL, SPiCE, CMMi, CObIT, ISO 17799, BS 7799, MDA, Six Sigma et IT Gouvernance*. Management et informatique. Hermes Science Publications, 2006. 152

[20] CAWSEY, A., AND OF EDINBURGH. DEPT. OF ARTIFICIAL INTELLIGENCE, U. *Explaining the Behaviour of Simple Electronic Circuits*. No. 376 in D.A.I. research paper. University of Edinburgh, Department of Artificial Intelligence, 1988. 24

[21] CÉRET, E., DUPUY-CHESSA, S., AND CALVARY, G. M2flex: a process metamodel for flexibility at runtime. *7th IEEE Int. Conf. On research Challenge in Information Science RCIS'2013* (2013). 184

[22] CERET, E., DUPUY-CHESSA, S., AND GODET-BAR, G. Using software metrics in the evaluation of a conceptual component model. In *Research Challenges in Information Science (RCIS), 2010 Fourth International Conference on* (2010), pp. 507–514. 159

[23] CHEN, P. P.-S. The entity-relationship modeltoward a unified view of data. *ACM Trans. Database Syst. 1*, 1 (Mar. 1976), 9–36. 68

[24] CHERFI, S. S.-S., AKOKA, J., AND COMYN-WATTIAU, I. Conceptual modeling quality - from eer to uml schemas evaluation. In *Proceedings of the 21st International Conference on Conceptual Modeling* (London, UK, UK, 2002), ER '02, Springer-Verlag, pp. 414–428. 153

[25] CLANCEY, W. J. The epistemology of a rule-based expert system: a framework for explanation. Tech. rep., Stanford, CA, USA, 1981. 16

[26] COUTAZ, J. Pac, an object oriented model for dialog design. In *Proceedings Interact* (1987), vol. 87, pp. 431–436. 48

[27] DECKER, K., PANNU, A., SYCARA, K. P., AND WILLIAMSON, M. Designing behaviors for information agents. In *Autonomous Agents and Multiagent Systems/International Conference on Autonomous Agents* (1997), pp. 404–412. 19

[28] DELISLE, S., AND MOULIN, B. User interfaces and help systems: from helplessness to intelligent assistance. *Artificial Intelligence 18*, 2 (Oct. 2002), 117–157. 2

[29] DEMEURE, A., LEHMANN, G., PETIT, M., AND CALVARY, G. Enhancing interaction with supplementary supportive user interfaces (uis): meta-uis, mega-uis, extra-uis, supra-uis . . . In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems* (New York, NY, USA, 2011), EICS '11, ACM, pp. 333–334. 6, 259

[30] DITTMAR, A., GARCÍA FREY, A., AND DUPUY-CHESSA, S. What can model-based ui design offer to end-user software engineering? In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems* (New York, NY, USA, 2012), EICS '12, ACM, pp. 189–194. 215

[31] DIX, A., FINLEY, J., ABOWD, G., AND BEALE, R. *Human-computer interaction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998. 197

[32] DORATO, M., AND FELLINE, L. Scientific explanation and scientific structuralism, January 2010. 14

[33] DROMEY, R. G. A model for software product quality. *IEEE Trans. Softw. Eng. 21*, 2 (Feb. 1995), 146–162. 90

[34] DROMEY, R. G. Concerning the chimera. *IEEE Softw. 13*, 1 (Jan. 1996), 33–43. 90

[35] EISENSTEIN, J., AND RICH, C. Agents and guis from task models. In *Proceedings of the 7th international conference on Intelligent user interfaces* (New York, NY, USA, 2002), IUI '02, ACM, pp. 47–54. 29, 198

[36] ELMASRI, R., AND NAVATHE, S. B. *Fundamentals of Database Systems.* Benjamin/Cummings, 1989. 68

[37] FAVRE, J.-M. Foundations of meta-pyramids: Languages vs. metamodels – episode ii: Story of thotus the baboon1. In *Language Engineering for Model-Driven Software Development* (Dagstuhl, Germany, 2005), J. Bezivin and R. Heckel, Eds., no. 04101 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. 71

[38] FAVRE, J.-M. Foundations of model (driven) (reverse) engineering : Models – episode i: Stories of the fidus papyrus and of the solarus. In *Language Engineering for Model-Driven Software Development* (Dagstuhl, Germany, 2005), J. Bezivin and R. Heckel, Eds., no. 04101 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. 70

[39] FININ, T., FRITZSON, R., MCKAY, D., AND MCENTIRE, R. Kqml as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management* (New York, NY, USA, 1994), CIKM '94, ACM, pp. 456–463. 19

[40] FINNE, A. Towards a quality meta-model for information systems. *Software Quality Journal 19*, 4 (2011), 663–688. 96, 97, 236

[41] FOLEY, J., AND VAN DAM, A. *Fundamentals of interactive computer graphics.* The systems programming series. Addison-Wesley Pub. Co., 1982. 47

[42] FOWLER, J. Variant design for mechanical artifacts: A state-of-the-art survey. *Engineering with Computers 12*, 1 (1996), 1–15. 147

[43] GARCÍA FREY, A., CALVARY, G., AND DUPUY-CHESSA, S. Self-explanatory user interfaces by model-driven engineering. In *Proceedings of the second ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2010)* (2010), ACM Press, pp. 341–344.

[44] GARCÍA FREY, A., CALVARY, G., AND DUPUY-CHESSA, S. Self-explanatory user interfaces by model-driven engineering. In *Proceedings of the CHI'10 Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI'10)* (2010), pp. 1–4.

[45] GARCÍA FREY, A., CALVARY, G., AND DUPUY-CHESSA, S. Xplain: an editor for building self-explanatory user interfaces by model-driven engineering. In *Proceedings of the second ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2010)* (2010), ACM Press, pp. 41–46.

[46] GARCÍA FREY, A., CALVARY, G., AND DUPUY-CHESSA, S. Users need your models! exploiting design models for explanations. In *Proceedings of HCI 2012, Human Computer Interaction, People and Computers XXVI, The 26th BCS HCI Group conference (Birmingham, UK)* (2012).

[47] GARCÍA FREY, A., CALVARY, G., DUPUY-CHESSA, S., AND MANDRAN, N. Model-based self-explanatory UIs for free, but are they valuable? In *Proceedings of the 14th IFIP TC13 Conference on Human-Computer Interaction (INTERACT'13), 2-6 September 2013, Cape Town, South Africa* (2013), Springer.

[48] GARCÍA FREY, A., CÉRET, E., DUPUY-CHESSA, S., AND CALVARY, G. QUIMERA - toward an unifying quality metamodel. In *Congrès INFORSID'11 (Lille, France, May 2011), 6 pages.* (2011).

[49] GARCÍA FREY, A., CÉRET, E., DUPUY-CHESSA, S., AND CALVARY, G. QUIMERA: a quality metamodel to improve design rationale. In *Proceedings of the third ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2011)* (2011), ACM Press, pp. 265–270.

[50] GARCÍA FREY, A., CÉRET, E., DUPUY-CHESSA, S., CALVARY, G., AND GABILLON, Y. Usicomp: an extensible model-driven composer. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems* (New York, NY, USA, 2012), EICS '12, ACM, pp. 263–268. 176, 184

[51] GODET-BAR, G., RIEU, D., DUPUY-CHESSA, S., AND JURAS, D. Interactional objects: Hci concerns in the analysis phase of the symphony method. In *ICEIS (5)* (2007), J. Cardoso, J. Cordeiro, and J. Filipe, Eds., pp. 37–44. 159

[52] GOOD, N., SCHAFER, J. B., KONSTAN, J. A., BORCHERS, A., SARWAR, B., HERLOCKER, J., AND RIEDL, J. Combining collaborative filtering with personal agents for better recommendations. In *Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence* (Menlo Park, CA, USA, 1999), AAAI '99/IAAI '99, American Association for Artificial Intelligence, pp. 439–446. 19

[53] GRAESSER, A., AND MURACHVER, T. *Symbolic procedures of question answering. Chapter 2 in The Psychology of Questions by Graesser, A.C. and Black, J.B.* L. Erlbaum Associates, 1985. 24

[54] GREGOR, S., AND BENBASAT, I. Explanations from intelligent systems: theoretical foundations and implications for practice. *MIS Q. 23*, 4 (Dec. 1999), 497–530. 16, 17, 44

[55] HAYNES, S. R., COHEN, M. A., AND RITTER, F. E. Designs for explaining intelligent agents. *Int. J. Hum.-Comput. Stud. 67*, 1 (Jan. 2009), 90–110. 19, 20

[56] HEIDENREICH, F., JOHANNES, J., SEIFERT, M., AND WENDE, C. Closing the gap between modelling and java. In *Software Language Engineering*, M. van den Brand, D. Gaševic, and J. Gray, Eds., vol. 5969 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 374–383. 183

[57] HEMPEL, C. G., AND OPPENHEIM, P. Studies in the logic of explanation. *Philosophy of Science 15*, 2 (1948), 135–175. 13

[58] HERMJAKOB, U. Parsing and question classification for question answering. In *Proceedings of the workshop on Open-domain question answering - Volume 12* (Stroudsburg, PA, USA, 2001), ODQA '01, Association for Computational Linguistics, pp. 1–6. 22

[59] HORTON, W. *Designing and writing online documentation: hypermedia for self-supporting products.* Wiley technical communication library. Wiley, 1994. 197

[60] HUGHES, R. Bell's theorem, ideology, and structural explanation. In *Philosophical consequences of quantum theory: reflections on Bell's theorem* (1989), J. Cushing and E. McMullin, Eds., University of Notre Dame Press, pp. 195–207. 14

[61] HUSSMANN, H., MEIXNER, G., AND ZUEHLKE, D. *Model-driven development of advanced user interfaces.* Springer, Berlin, 2011. 5

[62] ISO. *International Standard ISO/IEC 14598-1: Information Technology - Software Product Evaluation - Part 1: General Overview.* Geneva, Switzerland, 1999. 87

[63] JACKSON, P. *Introduction to Expert Systems*, 3rd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998. 15

[64] JACOBSON, I. *Object-oriented software engineering: a use case driven approach.* ACM Press Series. ACM Press, 1992. 68

[65] JENNINGS, N. R. Agent-based computing: Promise and perils. In *In Proceedings of the 16th International Joint Conference on Artificial Intelligence* (1999), pp. 1429–1436. 19

[66] JOUAULT, F., ALLILAIRE, F., BÉZIVIN, J., KURTEV, I., AND VALDURIEZ, P. Atl: a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2006), OOPSLA '06, ACM, pp. 719–720. 181

[67] KITCHENHAM, B., AND PFLEEGER, S. L. Software quality: The elusive target. *IEEE Softw. 13*, 1 (Jan. 1996), 12–21. 88, 149

[68] KLEPPE, A., WARMER, J., AND BAST, W. *MDA Explained: The Model Driven Architecture : Practice and Promise.* The Addison-Wesley Object Technology Series. Addison-Wesley, 2003. 70

[69] KO, A. J., AND MYERS, B. A. Development and evaluation of a model of programming errors. In *Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments* (Washington, DC, USA, 2003), HCC '03, IEEE Computer Society, pp. 7–14. 108

[70] KUNZ, W., AND RITTEL, H. *Issues as Elements of Information Systems.* Center for Planning and Development Research, University of California at Berkeley, 1970. 147

[71] LASHKARI, Y., METRAL, M., AND MAES, P. Collaborative interface agents. In *In Proceedings of the Twelfth National Conference on Artificial Intelligence* (1994), AAAI Press, pp. 444–449. 19

[72] LEE, J. Extending the potts and bruns model for recording design rationale. In *Proceedings of the 13th international conference on Software engineering* (Los Alamitos, CA, USA, 1991), ICSE '91, IEEE Computer Society Press, pp. 114–125. 147

[73] LEE, J., AND LAI, K.-Y. What's in design rationale? *Hum.-Comput. Interact. 6*, 3 (Sept. 1991), 251–280. 104

[74] LEHNERT, W. *The process of question answering.* Yale., 1977. 25, 26

[75] LEHNERT, W. *The Process of Question Answering: A Computer Simulation of Cognition.* Artificial Intelligence Series. Erlbaum, 1978. 21, 25

[76] LEONDES, C. *Expert systems: the technology of knowledge management and decision making for the 21st century.* No. vol. 2 in Expert Systems: The Technology of Knowledge Management and Decision Making for the 21st Century. Academic Press, 2002. 15

[77] LEWANDOWSKI, A., LEPREUX, S., AND BOURGUIN, G. Tasks models merging for high-level component composition. In *Proc. of HCI'07* (Berlin, Heidelberg, 2007), Springer-Verlag, pp. 1129–1138. 115

[78] LIM, B. Y., AND DEY, A. K. Assessing demand for intelligibility in context-aware applications. In *Proceedings of the 11th international conference on Ubiquitous computing* (New York, NY, USA, 2009), Ubicomp '09, ACM, pp. 195–204. 31

[79] LIM, B. Y., AND DEY, A. K. Toolkit to support intelligibility in context-aware applications. In *Proceedings of the 12th ACM international conference on Ubiquitous computing* (New York, NY, USA, 2010), Ubicomp '10, ACM, pp. 13–22. 59, 61, 235

[80] LIM, B. Y., DEY, A. K., AND AVRAHAMI, D. Why and why not explanations improve the intelligibility of context-aware intelligent systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2009), CHI '09, ACM, pp. 2119–2128. 5, 31, 195

[81] LIMBOURG, Q., VANDERDONCKT, J., MICHOTTE, B., BOUILLON, L., AND LÓPEZ-JAQUERO, V. USIXML: a language supporting multi-path development of user interfaces. In *Engineering Human Computer Interaction and Interactive Systems*, R. Bastide, P. Palanque, and J. Roth, Eds., vol. 3425 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, pp. 134–135. 10.1007/11431879_12. 113

[82] LÓPEZ-JAQUERO, V., VANDERDONCKT, J., MONTERO, F., AND GONZÁLEZ, P. Engineering interactive systems. Springer-Verlag, Berlin, Heidelberg, 2008, ch. Towards an Extended Model of User Interface Adaptation: The Isatine Framework, pp. 374–392. 104

[83] MACLEAN, A., YOUNG, R., BELLOTTI, V., AND MORAN, T. Questions, Options, and Criteria: Elements of Design Space Analysis. *Human-Computer Interaction 6*, 3 (Sept. 1991), 201–250. 104

[84] MACLEAN, A., YOUNG, R. M., BELLOTTI, V. M. E., AND MORAN, T. P. Questions, options, and criteria: elements of design space analysis. *Hum.-Comput. Interact. 6*, 3 (Sept. 1991), 201–250. 46, 86, 148

[85] MAES, P. Agents that reduce work and information overload. *Commun. ACM 37*, 7 (July 1994), 30–40. 18, 234

[86] MARPLES, D., AND KRIENS, P. P (2001) the open service gateway initiative: an introductory overview. *In: IEEE Commun Mag. pp.* 176

[87] MAYBURY, M., AND WAHLSTER, W. *Readings in intelligent user interfaces.* Morgan Kaufmann Series in Interactive Technologies. Morgan Kaufmann Publishers, 1998. 18

[88] MCCALL, J. A., RICHARDS, P. K., AND WALTERS, G. F. Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality. Technical Report ADA049014, GENERAL ELECTRIC CO SUNNYVALE CALIF, Nov. 1977. 88, 149

[89] MCCALL, R. Phi: A conceptual foundation for design hypermedia. *Design Studies 12,* 1 (1991), 30–41. 147

[90] MCGUINNESS, D. L., GLASS, A., WOLVERTON, M., AND SILVA, P. P. D. A categorization of explanation questions for task processing systems. 29

[91] MCKEOWN, K. R. Text generation: Using discourse strategies and focus constraints to generate natural language text. Cambridge University Press. 23

[92] MCMULLIN, E. Structural explanation. In *American Philosophical Quarterly, Vol. 15, No. 2* (April 1978), N. Rescher, Ed., University of Illinois Press, pp. 139–147. 14

[93] MDA. Model driven architecture - a technical perspective. Tech. Rep. ab/2001-02-01, OMG, 2001. Architecture Board MDA Drafting Team Review Draft. 69

[94] MELLOR, S. *Mda Distilled: Principles of Model-Driven Architecture.* Addison-Wesley Object Technology Series. Addison-Wesley, 2004. 70

[95] MELLOR, S. J., CLARK, A. N., AND FUTAGAMI, T. Guest editors' introduction: Model-driven development. *IEEE Software 20* (2003), 14–18. 78

[96] MOLDOVAN, D. I., HARABAGIU, S. M., PASCA, M., MIHALCEA, R., GOODRUM, R., GIRJU, R., AND RUS, V. Lasso: A tool for surfing the answer net. In *TREC* (1999). 22

[97] MOORE, J. D., AND SWARTOUT, W. R. Explanation in expert systems: A survey. ISI Research Report, ISI/RR-88-228, Information Science Institute, University of Southern California, Los Angeles, CA, 1988. 15

[98] MORAN, T., AND CARROLL, J. *Design rationale: concepts, techniques, and use*. Computers, Cognition, and Work Series. Lawrence Erlbaum Associates, Incorporated, 1996. 147

[99] MYERS, B., HOLLAN, J., CRUZ, I., BRYSON, S., BULTERMAN, D., CATARCI, T., CITRIN, W., GLINERT, E., GRUDIN, J., AND IOANNIDIS, Y. Strategic directions in human-computer interaction. *ACM Comput. Surv. 28*, 4 (Dec. 1996), 794–809. 5

[100] MYERS, B. A., WEITZMAN, D. A., KO, A. J., AND CHAU, D. H. Answering why and why not questions in user interfaces. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems* (New York, NY, USA, 2006), ACM, pp. 397–406. 1, 5, 30, 52, 54, 195, 214, 235

[101] NECHES, R., SWARTOUT, W. R., AND MOORE, J. D. Enhanced maintenance and explanation of expert systems through explicit models of their development. *IEEE Trans. Softw. Eng. 11*, 11 (Nov. 1985), 1337–1351. 23

[102] NICHOLS, J., AND KANG, J.-H. Asking questions of targeted strangers on social networks. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work* (New York, NY, USA, 2012), CSCW '12, ACM, pp. 999–1002. 31, 234

[103] NICOLOSI, E., LEANING, M., AND BOROUJERDI, M. The development of an explanatory system using knowledge-based models. In *Proceedings of the 4th Explanations Workshop (KB Design group)* (1988), Manchester University, pp. 14–16. 26

[104] NIELSEN, J., AND MOLICH, R. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 1990), CHI '90, ACM, pp. 249–256. 1

[105] NILSSON, N. *Artificial Intelligence: A New Synthesis.* The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann Publishers, 1998. 15

[106] NORMAN, D. *The design of everyday things.* New York: Doubleday, 1990. 104

[107] NORMAN, D. A., AND DRAPER, S. W. *User Centered System Design; New Perspectives on Human-Computer Interaction.* L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1986. 105

[108] OF TECHNOLOGY), P. M. M. I., AND MALHOTRA, A. *Design criteria for a knowledge-based English language system for management: an experimental analysis.* 1975. 23

[109] OMG. MDA Guide Version 1.0.1. `http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf`, June 2003. 69, 70, 71, 74

[110] OMG. MDA Guide Working Page. `http://ormsc.omg.org/mda_guide_working_page.htm/`, 2007. [Online; accessed 5-July-2012]. 69

[111] ORGANIZATION, I. S. ISO 9126 information technology - software product evaluation - quality characteristics and guidelines for their use. Geneva, Switzerland, 1991. 91

[112] PAILLÉ, P., AND MUCCHIELLI, A. *L'analyse qualitative: en sciences humaines et sociales.* Collection U. Sciences Sociales. Armand Collin, 2003. 193

[113] PALANQUE, P., BASTIDE, R., AND DOURTE, L. Contextual help for free with formal dialogue design. 29

[114] PANGOLI, S., AND PATERNÓ, F. Automatic generation of task-oriented help. In *Proceedings of the 8th annual ACM symposium on User interface and software technology* (New York, NY, USA, 1995), UIST '95, ACM, pp. 181–187. 29, 113

[115] PATERNÒ, F., MANCINI, C., AND MENICONI, S. Concurtasktrees: A diagrammatic notation for specifying task models. In *Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction* (London, UK, UK, 1997), INTERACT '97, Chapman & Hall, Ltd., pp. 362–369. 29, 83

[116] PETERSON, J. *Petri net theory and the modeling of systems.* Foundations of Philosophy Series. Prentice-Hall, 1981. 29

[117] PFLEEGER, S. *Software engineering: theory and practice.* An Alan R. Apt Book Series. Prentice Hall, 2001. 89

[118] PILKINGHTON, R., TATTERSALL, C., AND HARTLEY, R. Instructional dialogue management. In *CEC ESPRIT* (1988), EUROHELP, p. 280. 26

[119] POMERANTZ, J. Question taxonomies for digital reference. *SIGIR Forum 38*, 1 (July 2004), 79–79. 13

[120] POTTS, C. A generic model for representing design methods. In *Proceedings of the 11th international conference on Software engineering* (New York, NY, USA, 1989), ICSE '89, ACM, pp. 217–226. 147

[121] PRIOR, M., AND PRIOR, A. Erotetic Logic. *Philosophical Review 64*, 1 (1955), 43–59. 13

[122] PURCHASE, H. C., AND WORRILL, J. An empirical study of on-line help design: features and principles. *Int. J. Hum.-Comput. Stud. 56*, 5 (Apr. 2002), 539–567. 5

[123] QIAN, K., FU, X., AND TAO, L. *Software Architecture and Design Illuminated.* Jones and Bartlett illuminated series. Jones & Bartlett Learning, 2009. 48

[124] RICCI, F., ROKACH, L., SHAPIRA, B., AND KANTOR, P. B., Eds. *Recommender Systems Handbook.* Springer, 2011. 33

[125] ROBINSON, W., AND RACKSTRAW, S. *A question of answers.* No. vol. 1 in Primary socialization, language and education. Routledge and K. Paul, 1972. 22

[126] ROBINSON, W., AND RACKSTRAW, S. *A question of answers.* No. vol. 2 in Primary socialization, language and education. Routledge and K. Paul, 1972. 22

[127] RUMBAUGH, J. *Object-oriented modeling and design.* Object-oriented modeling and design / James Rumbaugh. Prentice Hall, 1991. 68

[128] SCHMIDT, D. C. Guest editor's introduction: Model-driven engineering. *Computer 39* (2006), 25–31. 79

[129] SCHURZ, G. Scientific explanation: A critical survey. *Foundations of Science 1* (1995), 429–465. 12

[130] SEFFAH, A., KECECI, N., AND DONYAEE, M. Quim: A framework for quantifying usability metrics in software quality models. In *Proceedings of the Second Asia-Pacific Conference on Quality Software* (Washington, DC, USA, 2001), APAQS '01, IEEE Computer Society, pp. 311–. 94, 151

[131] SELIC, B. The pragmatics of model-driven development. *Software, IEEE 20*, 5 (sept.-oct. 2003), 19 – 25. 78

[132] SELLEN, A., AND NICOL, A. Human-computer interaction. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995, ch. Building user-centered on-line help, pp. 718–723. 28

[133] SERNA, A., CALVARY, G., AND SCAPIN, D. L. How assessing plasticity design choices can improve ui quality: a case study. In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems* (New York, NY, USA, 2010), EICS '10, ACM, pp. 29–34. 104

[134] SHNEIDERMAN, B. Direct manipulation for comprehensible, predictable and controllable user interfaces. In *PROCEEDINGS OF IUI97, 1997 INTERNATIONAL CONFERENCE ON INTELLIGENT USER INTERFACES* (1997), ACM Press, pp. 33–39. 19

[135] SHNEIDERMAN, B., AND PLAISANT, C. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 2010. 45, 48, 197

[136] SHNEIDERMAN, B., AND UNIVERSITY OF MARYLAND (COLLEGE PARK, M. H. I. L. *Universal Usability: Pushing Human-computer Interaction Research to Empower Every Citizen*. Computer science technical report series. Human-Computer Interaction Laboratory, Institute for Advanced Computer Studies, 1999. 5

[137] SOTTET, J.-S. *Méga-IHM : malléabilité des Interfaces Homme-Machine dirigées par les modèles*. PhD thesis, 2008. Thèse de doctorat Informatique préparée au Laboratoire d'Informatique de Grenoble (LIG), Université Joseph Fourier. 71

[138] SOTTET, J.-S., CALVARY, G., FAVRE, J.-M., AND COUTAZ, J. Megamodeling and metamodel-driven engineering for plastic user interfaces: Mega-ui. In *Human-Centered Software Engineering*. 2009, pp. 173–200. 215

[139] SPIEKER, P. Natürlichsprachliche erklärungen in technischen expertensystemen. dissertation. Tech. rep., 1991. 28, 29

[140] STEINBERG, D., BUDINSKY, F., PATERNOSTRO, M., AND MERKS, E. *EMF: Eclipse Modeling Framework (2nd Edition)*, 2 ed. Addison-Wesley Professional, Dec. 2008. 180

[141] SUKAVIRIYA, P., AND FOLEY, J. D. Coupling a ui framework with automatic generation of context-sensitive animated help. In *Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology* (New York, NY, USA, 1990), UIST '90, ACM, pp. 152–166. 29, 57, 235

[142] SWARTOUT, W. R. Xplain: a system for creating and explaining expert consulting programs. *Artif. Intell. 21*, 3 (Sept. 1983), 285–325. 16, 17

[143] SWARTOUT, W. R. Xplain: a system for creating and explaining expert consulting programs. *Artificial Intelligence 21*, N° 3 (Sept. 1983), 285–325. 23

[144] SWARTOUT, W. R., AND SMOLIAR, S. W. Ai tools and techniques. Ablex Publishing Corp., Norwood, NJ, USA, 1989, ch. On making expert systems more like experts, pp. 197–216. 16

[145] TAKEDA, K., INABA, M., AND SUGIHARA, K. User interface and agent prototyping for flexible working. *IEEE MultiMedia 3*, 2 (Mar. 1996), 40–50. 19

[146] TENNANT, H. R. Experience with the evaluation of natural language question answerers. In *IJCAI* (1979), B. G. Buchanan, Ed., William Kaufmann, pp. 874–876. 23

[147] TINTAREV, N. Explaining recommendations. In *User Modeling 2007*, C. Conati, K. McCoy, and G. Paliouras, Eds., vol. 4511 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007, pp. 470–474. 33

[148] TOMURO, N. Question terminology and representation for question type classification. In *COLING-02 on COMPUTERM 2002: second international workshop on computational terminology - Volume 14* (Stroudsburg, PA, USA, 2002), COMPUTERM '02, Association for Computational Linguistics, pp. 1–7. 26

[149] USIXML. The UsiXML Language. `http://usixml.eu/about-the-project`, 2012. [Online; accessed 10-July-2012]. 85

[150] VALLEY, K. Explanation generation in an expert system shell. University of Edinbugh. Edinburgh. EH1 1HN. 1988. 26

[151] VAN MELLE, W., SHORTLIFFE, E. H., AND BUCHANAN, B. G. Emycin: A knowledge engineer's tool for constructing rule-based expert systems. In *Rule-based expert systems*. Addison-Wesley, 1984. 16

[152] VANDERDONCKT, J. *Guide ergonomique des interfaces homme-machine*. 1994. 99

[153] VERMEULEN, J., VANDERHULST, G., LUYTEN, K., AND CONINX, K. Pervasivecrystal: Asking and answering why and why not questions about pervasive computing applications. In *Proceedings of the 2010 Sixth International Conference on Intelligent Environments* (Washington, DC, USA, 2010), IE '10, IEEE Computer Society, pp. 271–276. 30, 54, 55, 113, 214, 235

[154] WATSON, A. H., MCCABE, T. J., AND WALLACE, D. R. Special publication 500-235, structured testing: A software testing methodology using the cyclomatic complexity metric. In *U.S. Department of Commerce/National Institute of Standards and Technology* (1996). 159

[155] WEXELBLAT, R. The confidence of their help. In *Proceedings of the AAAI'88 Workshop on Explanation* (1988), pp. 80–82. 24

[156] WICK, M. R., AND SLAGLE, J. R. An explanation facility for today's expert systems. *IEEE Expert: Intelligent Systems and Their Applications 4*, 1 (Mar. 1989), 26–36. 108

[157] WIKIPEDIA. Service (Systems Architecture) - Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/Service_(systems_architecture). [Accessed: Mar. 20, 2012], 2012. 176

Glossary

μ7 Concept that refers to seven different dimensions of user interfaces: multi-device, multi-user multi-culturality/linguality, multi-organisation, multicontext, multi-modality and multi-platform. *Glossary:* UsiXML

API An Application Programming Interface (API) is a particular set of rules and specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API. 258

Context of use is defined as the trio <user, platform, environment>. *Glossary:* Plastic UI

Plastic UI is a User Interface that is able to dynamically adapt to the context of use while preserving usability. *Glossary:*

Supportive User Interface A supportive user interface (SUI) exchanges information about an interactive system with the user, and/or enables its modification, with the goal of improving the effectiveness and quality of the user's interaction with that system [29]. 6, *Glossary:* SUI

User Interface refers to the graphical, textual and auditory information the program presents to the user, and the control sequences (such as keystrokes with the computer keyboard, movements of the computer mouse, and selections with the touchscreen) the user employs to control the program. 14

UsiXML The USer Interface eXtensible Markup Language is a XML-compliant markup language that describes the UI for multiple contexts of use such as Character User In-

terfaces (CUIs), Graphical User Interfaces (GUIs), Auditory User Interfaces, and Multimodal User Interfaces. It supports the concept of μ7. see. 258

Acronyms

API Application Programming Interface. *Glossary:* API

ER Entity-Relationship. 68

ISO International Organization for Standardization. 91

KBS Knowledge-Based System. 16

MBE Model-Based Engineering. 68

MDA Model-Driven Architecture. 68, 69

MDD Model-Driven Development. 68

MDE Model-Driven Engineering. 68

Model Driven UI Model-Driven User Interface. 203

OMG Object Management Group. 69

SUI Supportive User Interface. 6, *Glossary:* Supportive User Interface

UI User Interface. 1, 4, 5, 14, 203, *Glossary:* User Interface

UML Unified Modeling Language. 68, 69

UsiXML USer Interface eXtensible Markup Language. *Glossary:* UsiXML