PRISM: A Life Cycle to Align User Interface and Software Developments based on a Linguistic Perspective

Towards a User Interface Linguistic Paradigm

By Iyad Khaddam

A dissertation submitted in fulfilment of the requirements
for the degree of
Doctor of Philosophy in Engineering Sciences
of the Ecole Polytechnique de Louvain
Université catholique de Louvain

Committee in charge:
Prof. Jean Vanderdonckt, Université catholique de Louvain, Advisor
Prof. Christophe Kolski, Université de Valenciennes, France, Examiner
Dr. Costin Pribeanu, National Institute for Research and Development in Informatics, Romania, Examiner
Prof. Vivian Motti, George Mason University, Reader
Dr. Jorge Luis Perez Medina, Université catholique de Louvain, Reader

18 November 2016

A popular quote misattributed to Albert Einstein says:
"The definition of insanity is doing the same thing over and over and expecting different results."

In this thesis, I tried to be as wise as I can.

إهداء

إلى من لا زلت أرى نجاحي بعيونكما، إلى أبي وأمي.

To whom I see my success in their eyes. To my father and my mother.

إلى مسكن الروح وهناء القلب. إلى شريكة المغامرة والتعب والنجاح. إلى هناء زوجتي وصديقتي. احترت لأُهديك أم تهدينني؟ فلولا وجودك معي لما كان شيء من هذا.

To the home of my soul and to my partner in success. To Hanaa my wife and my friend. I wonder who should acknowledge the other? because without you everything here was impossible.

إلى الفسحة الجميلة في هذا العالم المتعب, إلى منبع النور في آخر المشوار زينة ونايا وحسن... كانت مغامرة صعبة لكنكم اجتزتموها بنجاح فأوصلتموني إلى النجاح. هي البداية...

To the light at the end of the journey, to my kids Zeina, Naya and Hasan. It was a hard adventure for you as well, but your success led to mine. It is a start…

إلى صاحب المشاعر الرهيفة التي تحيطني أينما حللت. إلى أخي كنان.

To the man with the sensitive feelings that surround me everywhere. To my brother Kinan.

إلى من تزيد محبتهم على قدر الغربة , إلى أخوتي خلود وديمة وهبة.

To my sisters Khouloud, Dima and Heba. The more we are far away, the more our feelings are intensified.

إلى أخي الكبير وائل الذي بفضله بدأت هذه المغامرة ثم انتهت نجاح، لكن فيض مشاعره دائم لا ينضب.

To the big brother Wael, without whom this adventure would have never started. Your feelings are surrounding us all the time.

إلى النسمات الشرقية الرقيقة العائمة في سماء أوروبا فلا تقبل أن تحط على أرضها ولا تقبل أن تغادرنا فهي تعرف مكانها. إلى جومو وهديل وميس.

To those delicate oriental winds flying in the sky of Europe, but refuse to land except in our hearts. To Jouman, Hadeel and Mais.

Acknowledgements

Firstly and foremost, my warm thanks to my advisor Jean Vanderdonckt who played the role of the wise man, whose words proof their wisdom to the ignorant trainee at the end of the journey.

My thanks also go to Prof. Christophe Kolski, Dr. Costin Pribeanu, Prof. Vivian Motti and Dr. Jorge Luis Perez Medina for accepting to be part of my jury and for dedicating from their time to examine, read and comment on the thesis. Your fruitful feedback enhanced the quality of this thesis very much.

I also wish to thank my colleagues both at Louvain Interaction Lab (LILab) and Louvain School of Management. To the open-minded Pascal who introduced me to a different culture through countless discussions and a lot of patience from his part. To Nesrine, Mathieu and Samedi who helped me sharpen the ideas in this thesis through a lot of fruitful discussions, even on creasy ones. To Jeremie, Hugo, Vivian, François, Diane, Hassan, Jorge, Suzanne and Efrem for both academic and friendship experiences. My thanks also go to Professors Marco and Manuel for their support and the unique teaching assistance experience.

To all the families with whom my family felt like home. My warmest thanks go to the Vanderdonckt family and the Zen family with whom we shared unforgettable moments and a unique cross-cultural Human-to-Human Interaction.

Lastly but not last, I wish to thank my friend Rachid Gherbi for his support and encouragement during this whole adventure.

Abstract

While Software Engineering (SE) is primarily concerned with developing software products, Human Computer Interaction (HCI) is primarily concerned with the User Interface (UI) of these products striving for its quality, such as usability. Therefore, integrating both disciplines is key to develop usable software products. The effort to align activities from both domains started since HCI emerged as a discipline in the early 80s. Yet, such an alignment proved to be a hard goal to reach. Consequently, valuable HCI contributions and models are not considered by the software industry as they could be.

In order to address the aforementioned challenge, this thesis analyzes different approaches to align HCI and SE in order to identify the root cause that is hampering the alignment effort. It follows a Root Cause Analysis methodology that helps, not only to identify the root cause, but also to propose a solution for it.

This thesis suggests the linguistic perspective as an alternative to common UI development which does not foster creating a concrete UI during the early phases (such as analysis), while the linguistic perspective enables this possibility by materializing linguistic levels (i.e., goal, pragmatic, semantic, syntactical, lexical, alphabetical) that are already analyzed into a partial yet concrete UI.

This change of perspective implies several changes to other HCI concepts which are discussed in this thesis: how a user interface can be developed from a linguistic perspective. For this purpose, this thesis proposes a linguistic UI development language (the Prism Programming Language). Moreover, it explores modeling approaches from the linguistic perspective, proposes a linguistic modeling framework and instantiates a graphical UI linguistic model from it. Furthermore, it develops the Prism Development Life Cycle (Prism-DLC) that can be instantiated in different software DLCs, such as agile methods. The Prism-DLC allows aligning SE and HCI in the right way to develop a usable software.

By fostering different way of thinking, this thesis steps forwards towards a shift in the UI paradigm by fostering a different way of thinking. The UI linguistic paradigm to common UI paradigm is like object oriented programming to procedural programming.

Keywords: User Interface, Development Life Cycle, Perspective, Linguistic Modeling, Linguistic Classification, Prism.

Table of Contents

Table of Figures

Table of tables

Acronyms

AIO	Abstract Interaction Object
AUI	Abstract User Interface
CTT	Concur Task Trees
CRF	Cameleon Reference Framework
CUI	Concrete User Interface
FUI	Final User Interface
GUI	Graphical User Interface
HTA	Hierarchical Task Analysis
HCI	Human Computer Interaction
IKIWISI	I cannot tell you, but I'll Know It When I See It
MDE	Model-Driven Engineering
SDLC	Software Development Life Cycle
SE	Software Engineering
SLR	A Systematic Review of the Literature
TDSC	Task Decomposition Stopping Criteria
UI	User Interface

Chapter 1 Introduction

Human-Computer Interaction (HCI) is a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them [ACM1994]. Software Engineering (SE) is the discipline that is concerned with the development of software systems. To enable developing usable[1] software products, both disciplines should be considered simultaneously.

Over the time, HCI established different approaches to address usability in products. A huge body of knowledge on usability guidelines has been accumulated. Different methods have been developed to gather usability requirements for a project (like the task and user models), none-the less, to analyze the impact of various contexts of use on usability. On the other hand, SE developed different methods to efficiently and effectively develop software products.

Integrating usability in the software development requires integrating HCI activities in the software development life cycle, which we call **aligning** activities in both disciplines. Such an alignment would mainly answer the question: how to develop a software product while considering usability throughout the development process of the software?

Usability is addressed mainly in the development of the User Interface (UI). Consequently, the integration of HCI and SE can be refined to the alignment of the UI development in the software development life cycle.

For some readers (especially from SE background), this might be confusing, as the common perspective to develop a UI is to consider it as part of the late design phase in the software development life cycle. This common perspective already aligns UI development activities with the design and the implementation phases. In fact, this common perspective is limiting to align HCI and SE activities because it prevents alignment before the design phase in SE.

[1] Usability is the extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use [ISO9241].

Prism re-defines UI activities in a different way than commonly done. It adopts a linguistic model of interaction to define UI development activities, which allows extracting activities at the analysis phase and consequently, establishes a UI development process that can be integrated in software development life cycles since the beginning of a project. In that regard, Prism enables the integration of usability requirements since the early beginning of the project development life cycle.

1.1 Motivation

The User Interface (UI) is important to the software product. It is the only tangible part of the product to the user. Therefore, it is the part that is subject to main critics, mainly on usability.

Development of UIs requires a considerable effort in the software development life cycle. According to an old study from the 90s, it occupies about 44% of the total effort to develop the complete software [Myers2000]. This value would look modest nowadays, considering the increased complexity in contexts of use, platforms and interaction styles from the time of conducting the study [Petrasch2007].

The effort on the UI development is spent on addressing usability issues in the UI. Aligning HCI activities with software development ones reduces the effort to address usability issues, by considering usability since the early beginning of the project.

Besides this, HCI has many contributions to the UI development, that when integrated in the development of a software, may significantly enhance usability.

1.2 Contextualization

Aligning the UI development in SE is an old problem. In the 70s, the software engineering faced a crisis due to the use of the slow and unreliable waterfall development process. Software Human Factors also faced a crisis: it was positioned at the late detailed design phase of the waterfall [Carroll2003]: the design of the User Interface (UI). Thus, it was involved after main design decisions had been taken. It was limited to cosmetic differences in software products. The identification of this crisis led to the emergence of HCI as a multi-disciplinary domain to start the development of the product since the beginning with consideration

of human factors.

Nowadays, the problem is not yet solved because HCI methods are not commonly adopted in SE. SE is evolving in its own way, while HCI is evolving with models and methods that are, unfortunately, not adopted by the industry [Hutch2011, Motti2013].

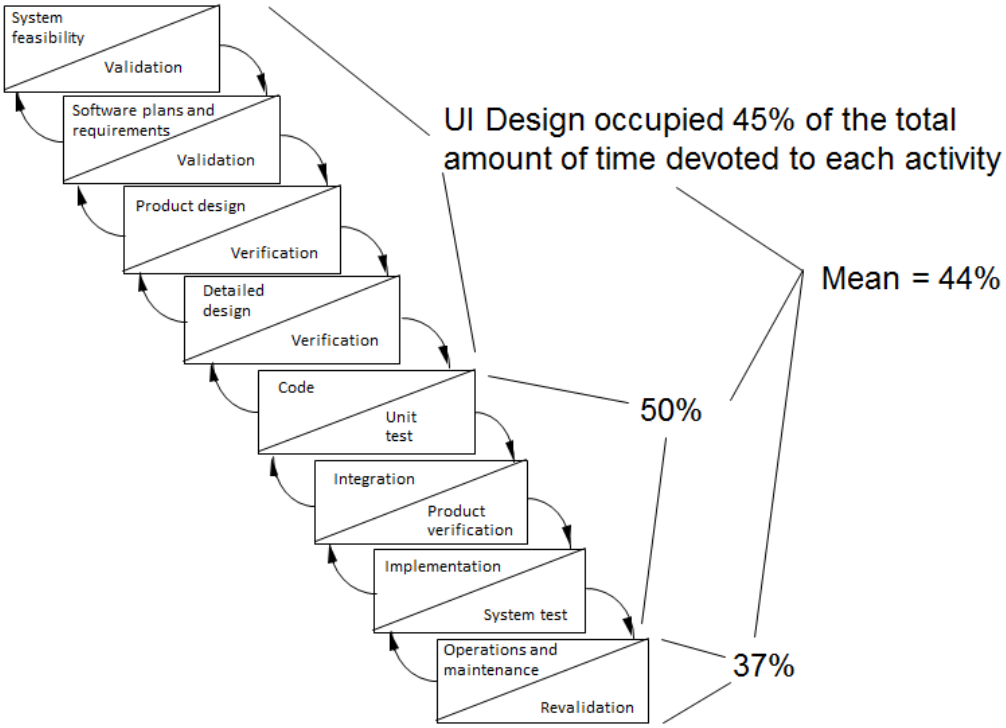

Figure 1.1 The distribution of UI design costs in a software project [Myers2000].

Anyway, we should note that SE tends to use HCI findings, but not HCI development models and methods. It is common to have a user experience role (or an equivalent) in software projects to advice on usability requirements, but not to apply HCI development methods. SE integrates HCI as a knowledge base of expertise, with less consideration of HCI development methods and models.

1.2.1 The UI in Software Engineering

The UI development is part of the Software Development Life Cycle (SDLC), but the relation between the development of the UI and the

software product is a bi-directional one, as depicted in figure 1.2:

1- The product design impacts the UI. We cannot build a UI independently of the product. Subtle information is required on the design and on the functionality of the product before designing the UI.
2- The UI impacts the software development. Users and stakeholders are often, during the early stages of the project, unable to express and describe what their requirements really are. This is known as the IKIWISI syndrome [Boehm2000], short for: *I cannot tell you, but I'll know when I see it*. Therefore, the UI may help the user to describe requirements after seeing and/or using the UI.

HCI helped in addressing this bi-directional relation. Figure 1.3 is a simple model of what we learnt in HCI and SE over the time. The software development is not sequential, and human factors play an important role in this development. The model in figure 1.3 depicts this bi-directional relation by introducing a new dimension to the development life cycle: the human factors dimension. This figure shows main generic software development activities that are followed in most development life cycles. It also shows that the UI resulting from the software development may not satisfy the user's expectations (mainly on usability), which are represented on the human factors axis as a level of knowledge. It also illustrates the gap in the analysis of human factors.

The model in figure 1.3 also helps in understanding how different SDLCs address the human factors problem. For instance, the waterfall model would fail if it does not start from the top right corner (start with a high level of knowledge on the human factors axis pertinent to the project). Iterative SDLCs fulfill the human factors analysis gap iteratively: we learn on these factors after each iteration (the full stack of development activities) until we reach the desired level of satisfaction for the client.

In software engineering, contacting the user to get feedback on the UI is the key to address the human factors crisis. The more effective contact with the user, the lower risk we have. Agility[2] is a main trend in SE to shorten the time needed to complete the full stack of development activities, and consequently, get the user feedback earlier.

[2] See the agile manifesto at the link : http://agilemanifesto.org/

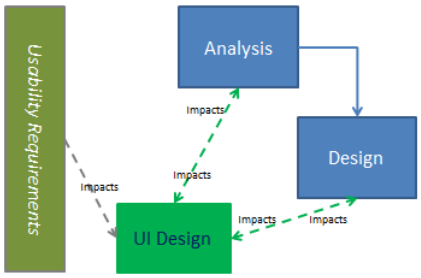

Figure 1.2 The bi-directional relation between the UI and the software development

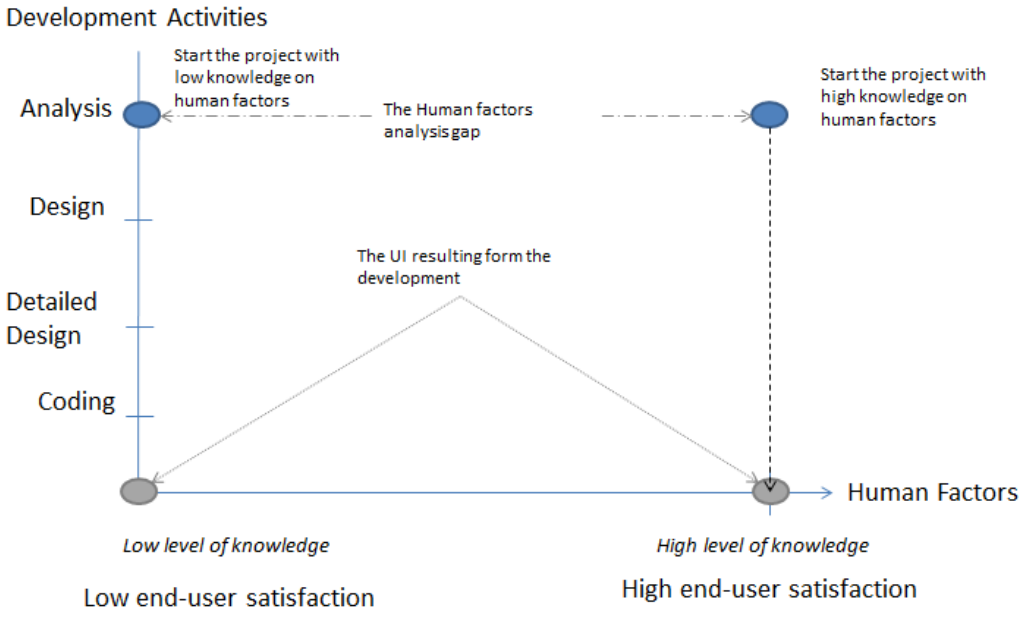

Figure 1.3 A model of the software development depicting the degree of knowledge on human factors to meet the user's expectations.

1.2.2 HCI and the UI Perspective

Human Computer Interaction (HCI) emerged as a discipline after the Human-Factors crisis, with the aim to integrate multiple fields of research. It encompasses linguistics, anthropology, philosophy, psychology and computer science [Carroll2003] among other fields and disciplines.

This diversity of fields constituting the HCI discipline impacted its contributions as well. The category of contributions concerned in this thesis is the one that impacts the development of usable UIs. This category focuses on creating methods and tools (mainly using modeling approaches) to develop usable UIs.

One of the valuable contributions of HCI is the way to develop the UI; the perspective to the UI development. HCI created new perspectives to the UI development, like starting the UI development from a task or a user model, in addition to employing different models for the UI, on different levels of abstractions. These contributions impacted the way to define and order UI development activities towards a usable UI.

UI development activities are defined after the perspective. Let's explain through two examples how activities are defined differently when the UI development perspective changes.

A UI developer; influenced by traditional UI development tools, may describe the UI development activities based on her/his experience with that tool. WYSIWYG (What You See Is What You Get) tools define UI development activities as: drag and drop UI widgets on the screen, then relate these UI elements to the code behind (the behavior). This perspective is depicted in figure 1.4 on top.

Modeling tasks is an activity that is not noticed from the UI developer's perspective in figure 1.4 on the top. The task-model perspective identifies a different set of activities to develop the UI, which starts with task model activities.

The HCI perspective is promising to the alignment because it allows starting the UI development since the early phases of the project, and thus opens the possibility to address human factors early. On the other side, the SE perspective to the UI development does not allow addressing human factors on the UI before the design phase, which prevents addressing human factors at the early phases.

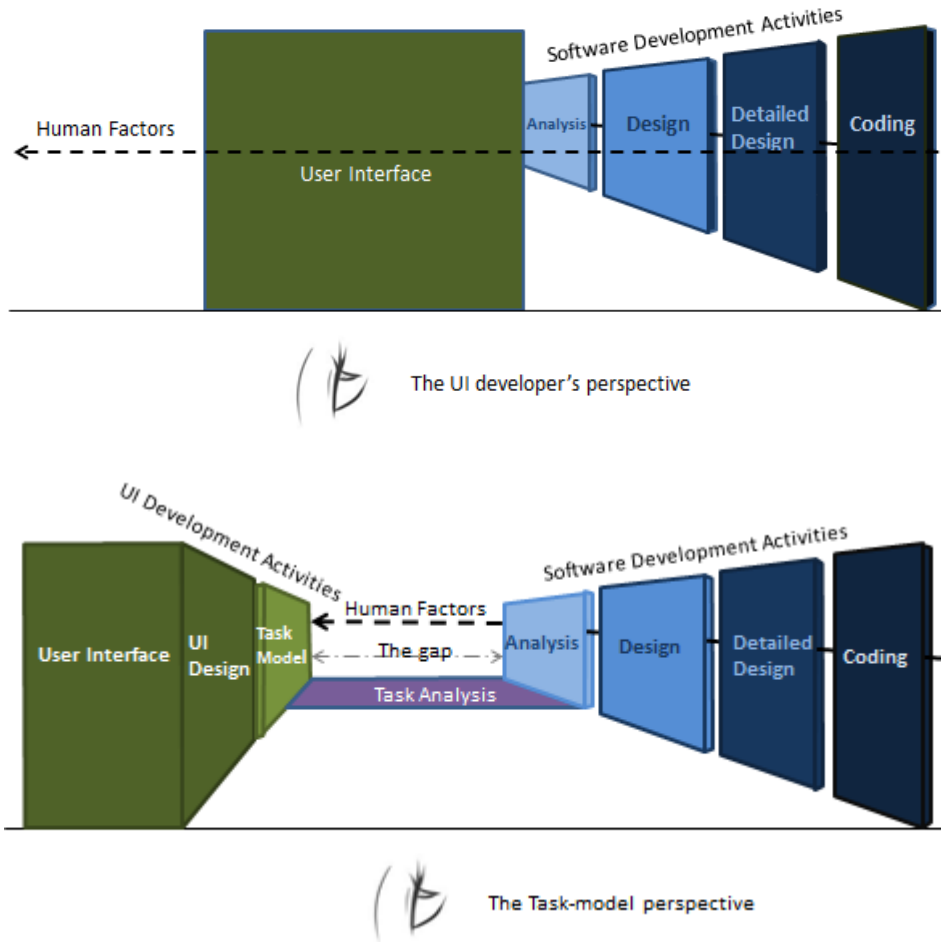

Figure 1.4 Perspectives to the UI development.

1.3 A Different Perspective to UI development

1.3.1 Nielsen's Virtual Protocol for Interaction

In 1986, Jacob Nielsen published an article [Nielsen1986] introducing his virtual protocol of interaction between the human and the computer. The protocol consists of seven levels of interaction that are decomposed after linguistic criteria: goal, pragmatic, semantic, syntactical, lexical, alphabetical and physical. The idea was considered a powerful one at that time by the community. Nielsen aimed at analyzing the interaction be-

tween the human and the machine from a linguistic point of view to provide a better understanding of the domain.

The original idea of Nielsen lies in expressing the interaction according to a linguistic classification benefiting from the following features:

1) Decomposition into linguistic layers: the interaction is decomposed into layers.
2) Communication between layers: each layer has two communication interfaces with other layers: an analyser interface with the upper layer, and a realizer interface with the lower one.
3) Concept Univocal Separation: each concept is univocally located in one and only one layer. This is the basis for the principle of separation of concerns [Dijkstra1976], [Parnas1972].
4) Layer Coverage: any interaction activity could be expressed with the full stack of layers

Nielsen explains that the communication between the human and the machine virtually happens at any level. In reality, it is happening on the physical level, as depicted in figure 1.5.

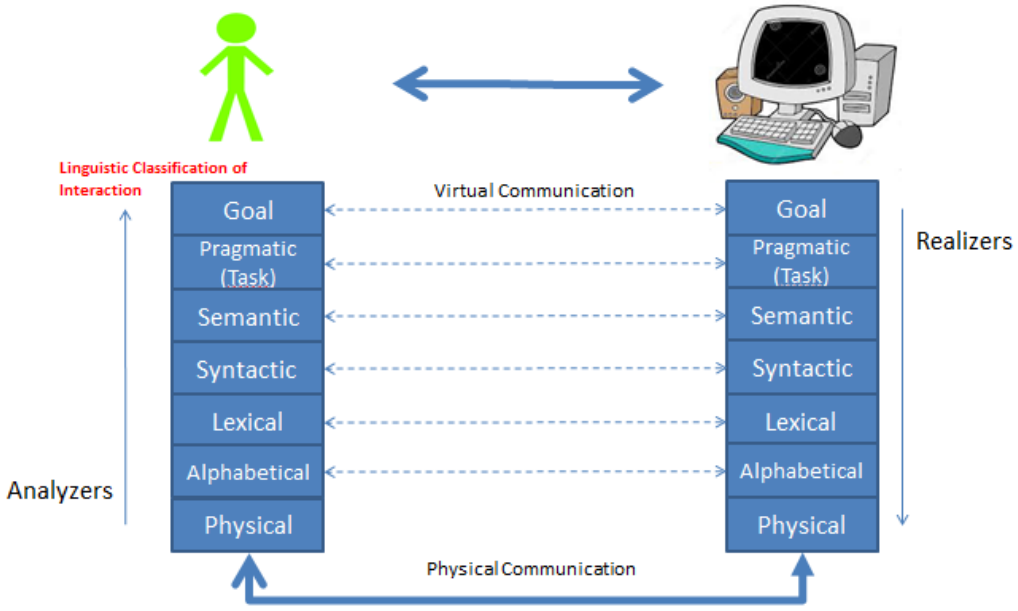

Figure 1.5 Overview of Nielsen's Virtual Protocol.

1.3.2 Towards a Linguistic Perspective

Nielsen's virtual protocol provides an opportunity to start the UI development since the early phases of the development. The interaction can be virtually carried out at any level, including the task level. This assumes a UI at each level to enable the user to interact with the system at that level. Therefore, we can expect a UI at the task level; that is refined later at the semantic level. This refinement is repeated until we reach the lowest level where the final concrete UI is used in the interaction.

Nielsen's linguistic model for interaction suggests a way to align the UI development and the SE based on a new perspective to the UI development that is more refined than the task model perspective (depicted in figure 1.4).

From such a linguistic perspective, the UI would be divided into a set of UIs, that when put all together, would result in a final and complete UI. We call the UI at the task layer: the "Task UI". The Semantic UI is a refined UI from the Task UI that is used to interact at the semantic layer. The same convention applies for all other layers.

The UI at the upper layers would provide a conceptual mapping with high-level UI concepts, while the UI at lower layers would provide the needed mapping to the perceptual world on a screen.

1.3.3 Linguistic Alignment of UI and SE activities

By adopting the linguistic perspective, the UI development process is better structured. The UI is developed iteratively at each linguistic layer to reach the final UI. Compare this structure to the common UI development perspective in SE, where the UI is developed as a step in the design phase. This structure is more detailed than the task model perspective, which can be fruitful to enhance the alignment of UI development with SE activities.

With such a detailed structure, we can imagine different possible methods to align UI development and SE activities. Thanks to the layer coverage feature of the linguistic perspective, we expect to have UI activities repartitioned/distributed on different levels of abstractions. On the other side, SE development activities cover all aspects of system development from the abstract level to the concrete one. **Aligning SE activities and UI activities** (defined from the linguistic perspective) **is a matter**

of finding the right ordering of activities at the same level of abstraction.

The alignment definition above focuses on the method to develop the UI as part of SE activities. A broad overview of such a method is depicted in figure 1.6. In this figure, we notice that a usable UI is developed at the end of the development activities. This UI is developed following a step-wise, iterative method that is synchronized with SE activities, with respect to repartitioning/distribution of activities per level of abstraction.

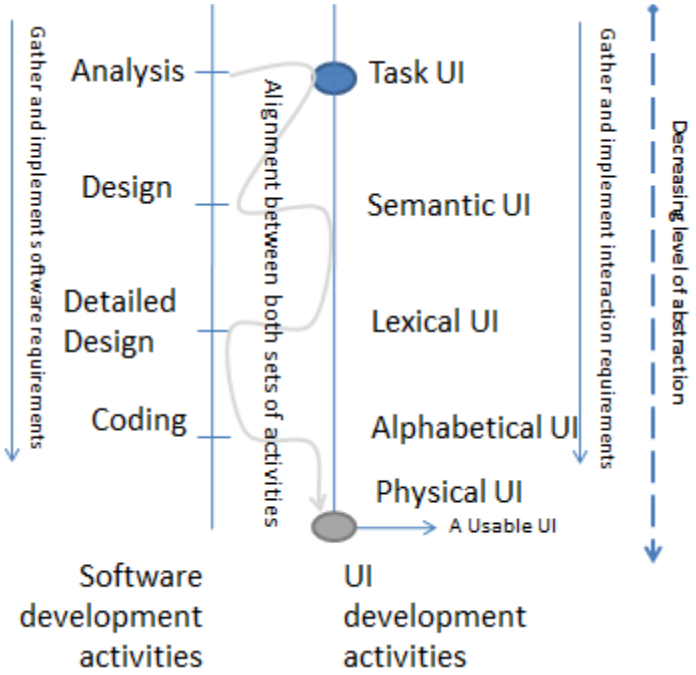

Figure 1.6 A broad view of a possible alignment method between UI and SE activities from a linguistic perspective.

1.3.4 The prism metaphor

The linguistic perspective classifies UI development activities on linguistic levels. This linguistic classification decomposes UI concepts and activities univocally on linguistic levels, which resembles the effect of a prism (in optics) on the visible light.

A prism in optics is an instrument to separate the white color into the

spectrum colors. The spectrum here is the classification of visible lights.

We look at the linguistic classification as an instrument to classify UI concepts and activities. We call this instrument the Linguistic Prism by anomaly to the prism in optics. The prism metaphor illustrates the univocal distribution of UI concepts per linguistic levels.

1.4 Thesis

1.4.1 Thesis Statement

The current perspective to the UI development, adopted in HCI and SE, is not appropriate to address human factors issues, because a concrete UI cannot be produced since the early beginning of the software development. A perspective to the UI that allows producing concrete UIs since the early beginning of the project development is needed in order to address usability issues in the software development. Such a perspective enhances integration of usability in the software product development.

Therefore, we defend the following statement:

A linguistic model of interaction structures the development of a concrete UI since the analysis phase and refines it on subsequent phases until the final concrete UI is reached. This induces a UI development life cycle with phases aligned to software development ones.

The linguistic model of interaction is Nielsen's Virtual Protocol for interaction. We aim at using Nielsen's model to **structure the development** of UIs. The promise of Nielsen's model is to enable the development **of concrete UI since the analysis phase**. We called this UI as the task UI, which is **refined on subsequent phases until the final concrete UI is reached**. Therefore, the Task UI is the part of the final concrete UI that is identified and developed at the task layer.

With this structure to the UI development, we can establish a **UI development life cycle** that starts from the analysis phase. Activities defined in this development life cycle can be **aligned** with activities as defined in a **software development life cycle**.

We achieve our goal by:

1- **The Linguistic Perspective:** Transform the linguistic model into a UI perspective that defines the UI development activities on several layers. This would establish a linguistic classification of UI development activities.
2- **A Linguistic UI model:** a model-based approach to develop UIs that follows the linguistic perspective.
3- **The Prism Software Development Life Cycle:** Align the linguistic UI development with software development by aligning the activities of both appropriately.

1.4.2 Scope

In this thesis, we focus on Graphical User Interfaces (GUIs) as our main concern. Extending our results and findings on other types of UIs is a work for the future.

Besides this, we also limit our scope to GUIs of Information Systems (ISs). Extending our findings and results to other types of applications is a work for the future.

The targeted audiences of this thesis are mainly HCI researchers and methodologist who are interested in new methods to develop the UI from a different perspective. Software engineers are also among the audience who might benefit from the induced prism development life cycle to explore new opportunities to integrate HCI findings in the software development life cycle.

This thesis is interested in defining a new method to develop the GUI. This method includes establishing a new perspective to the GUI development, based on the linguistic perspective and a linguistic classification of GUI concepts and artifacts on different linguistic levels.

The linguistic GUI development can be performed following different approaches, like a manual approach or a model-based one. Model-based approaches are of special interest for the HCI community. Therefore, we focus on model-based approaches as the approach to develop GUIs from the linguistic perspective. In that, we establish a linguistic modeling approach, which employs a model on each linguistic level. Linguistic models on each linguistic level constitute together a GUI linguistic model.

Model-based approaches are based on three pillars: Model, Method and

tool. The GUI Linguistic model requires a method to fulfill a specific requirement. The targeted requirement in this thesis is to align the GUI development with the software development activities. Therefore, the method to enact the GUI linguistic model is presented in the form of a development life cycle that aligns GUI development activities and software engineering ones. We call this method: the Prism DLC. However, this method can be adapted to meet the needs of a specific software development life cycle. Besides, different methods can be developed, based on Prism-DLC, to fulfil other requirements like operationalization of usability guidelines from a linguistic perspective, or adaptation of the GUI from a linguistic perspective among other requirements.

The GUI linguistic model requires a development tool. The tool developed in this thesis is a programming language that is called the Prism Programming Language (PrismPL). The PrismPL defines a set of statements per linguistic level. Each set of statements enable building the model on that linguistic level. Therefore, the PrismPL is the tool to develop the GUI linguistic model. It consists of a sub-tool at each linguistic level that enables developing the model on the linguistic level.

1.4.3 Definitions

System (Software) Development Life Cycle (SDLC) is "a conceptual model used in project management to describe the stages involved in a system development project from an initial feasibility study through maintenance of the completed application" [Royce1970].

A UI Perspective: the underlying perception of the UI domain in the mind of the domain practitioners that allow defining development activities.

Alignment of UI and software developments: Establish a well-defined interface between software development activities and the GUI ones. Software development activities refer to activities defined in generic development phases: the analysis, design, coding and testing.

Definitions concerning Software Activities:

Analysis: Requirements are the "capabilities and conditions to which the system must conform" [Jacobson1999]. They can be functional (capture the intended behavior of the system) or non-functional (qualities of the system that cannot be directly implemented by technical means but are a

result of it).

The purpose of the analysis is:

- To model stakeholders' requirements: provide models to allow communication between stakeholders and developers.
- To build a domain problem: a representation of all relevant concepts or real-world entities in a particular domain of interest

Design: Concerned with the elaboration of a solution to the problem described during the analysis phase. It is usually split into two distinct levels [Weyns2004]:

- Architectural Design: aims to build the system architecture that satisfies both functional and non-functional requirements.
- Detailed Design: aims to build a model for detailed diagrams and descriptions of all elements defined in the systems architecture. This model can directly be taken as input for the implementation.

Implementation: is concerned with the component building from scratch or by composition. The output is an executable product.

Test: is concerned with the evaluation of software correctness, completeness, security and quality.

1.5 Methodology

The alignment between the UI development and the software engineering is a concern to both HCI and SE communities since the identification of the Human Factors Crisis. To address this problem differently, we need to reveal root causes behind the problem. Once a root cause is identified, we can propose a solution to solve it.

This thesis follows the root cause analysis methodology that is proposed by Okes [Okes2009] and depicted in figure 1.7. This methodology consists of two main phases: the *Find It* and the *Fix It* phases.

The *Find It* phase is adapted for the purpose of this thesis as in the following. The adaptation consists of a Systematic Review of the literature (SLR) in steps 2,3 and 4. We describe below how each step is performed in this thesis.

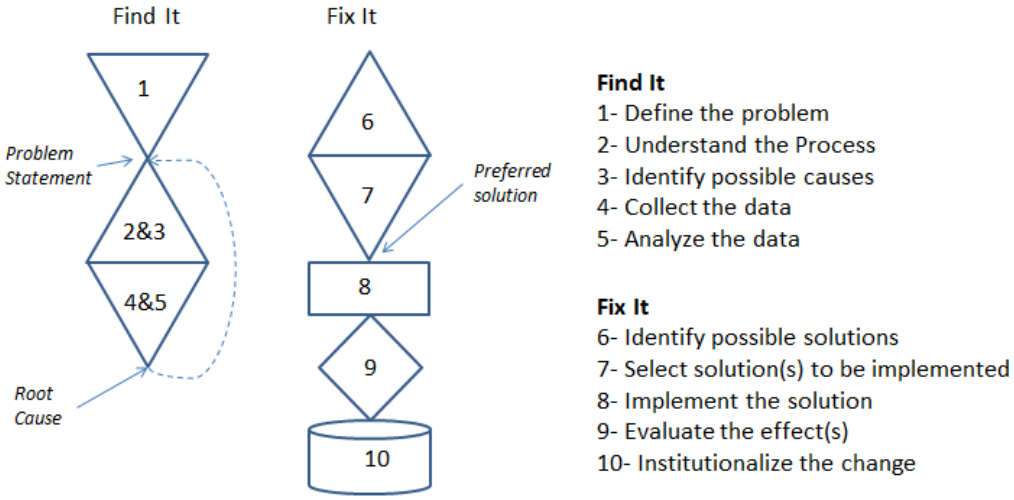

Figure 1.7 Methodology Overview

1- Define the problem: The alignment problem is defined in this chapter and the thesis statement is stated in section 1.4.1.
2- Understand the process: identify different approaches that address the alignment problem as explained in the literature. Identification of different approaches is based on a visit to the history aiming to understand how the alignment problem is addressed since it was identified. This is presented in chapter 2.
3- Identify possible causes: identify limitations in the approaches from step 2 as stated in the literature.
4- Collect the data: collect data from the literature on each approach. Organize these data in a form of conclusion statements to enhance the ability to perform step 5.
5- Analyze the data: reason on collected data, using conclusion statements from step 4, to identify the root cause behind the limitations for each approach.

The steps 3, 4 and 5 are repeated for each approach identified in step 2. The *Find It* phase is the concern of chapter 2.

The *Fix It* phase is adapted as follows:

6- Identify possible solutions: once the root cause is identified, we elicit requirements to fulfill by an acceptable solution. Requirements designate the solution space for the root cause. This is

performed at the end of chapter 2.

7- Select a solution to be implemented: We propose our solution to solve the root cause, which fulfills the requirements identified in step 6. The solution we propose is the linguistic perspective and an adapted version of the Nielsen's classification. This is performed in chapters 4 and 5.

8- Implement the solution: Implement a model-based approach based on the proposed solution in the step 7. We implement the solution by establishing a GUI linguistic model. This is the concern of chapters 5, 6 and 7. Each of these chapters follows a *Design, Implement and Validate* process for each model on a linguistic level. Validation is performed through a case study where we proof that a functional GUI can be developed from the proposed linguistic perspective, following a model-based approach, through iterations on each linguistic level.

9- Evaluate the effect: In step 2, we identified a list of shortcomings. In this step, we explain how these shortcomings are affected by solving the root cause.

10- Institutionalize the change: in this step, we put everything together in order to address the alignment problem. We explain how the linguistic perspective enables aligning GUI development and software development activities in a development life cycle. Chapter 9 is concerned with introducing the linguistic development life cycle that we call Prism-DLC.

1.6 Reading map

This thesis is organized as follows:

Chapter 1: defines the thesis statement, based on the definition and the understanding of the alignment problem between the software development and the UI development. It defines different perspectives to the UI development, and introduces a potential linguistic perspective. This chapter explains the methodology followed in this thesis, defines the terminology used and delimits its scope.

Chapter 2: is dedicated to perform steps 2, 3, 4, 5 and 6 of the methodology in 1.5. It identifies different approaches to address the alignment problem, identifies shortcomings as stated in the literature in each approach, gathers data on each approach and analyzes that data in order to

find the root cause behind identified shortcomings. Requirements are elicited based on the revealed root cause. This chapter is a hawk-eye analysis of the domain.

Chapter 3: introduces the linguistic perspective that is adapted from the linguistic model of Nielsen. We establish a linguistic classification of GUI concepts and activities, and introduce the linguistic prism: the linguistic classification of UI changes. This chapter partially performs the step 7 in the methodology.

Chapter 4: We demonstrate the linguistic perspective on a case study. We also discuss what is addressed from the list of requirements and how. This is a wrap-up chapter; it denotes a milestone in on the solution process. It sets the basis for following chapters. This chapter performs the step 7 in the methodology.

Chapters 5, 6 and 7: We introduce our GUI linguistic model in all these chapters. In each chapter, we introduce a model(s) on the linguistic level(s), implement and validate it. These chapters implement the proposed solution following a model-based approach. It implements the step 8 in the methodology.

Chapter 8: This is a wrap-up chapter. It evaluates the effect of the proposed solution on the shortcomings identified in chapter 2. This chapter sets the basis for the next and final chapter. It also denotes a milestone on the solution process. It implemented the step 9 in the methodology.

Chapter 9: We introduce the induced method Prism-DLC that aligns the UI development and the software development activities. This chapter denotes the final milestone and terminates the solution process. It implements the step 10 in the methodology.

Chapter 10: This chapter is dedicated to argue on the opportunity to establish a new paradigm, based on the linguistic perspective. It presents some examples to support our argument and illustrate the characteristics of the new paradigm.

Chapter 11: Concludes this thesis by summarizing its contributions. In addition, this chapter presents several possible extensions paths for future work.

Figure 1.8 depicts the story told in this thesis graphically. It depicts the 3 milestones we follow towards the solution. It also depicts the coverage

of chapters per phase on the analysis and solution processes.

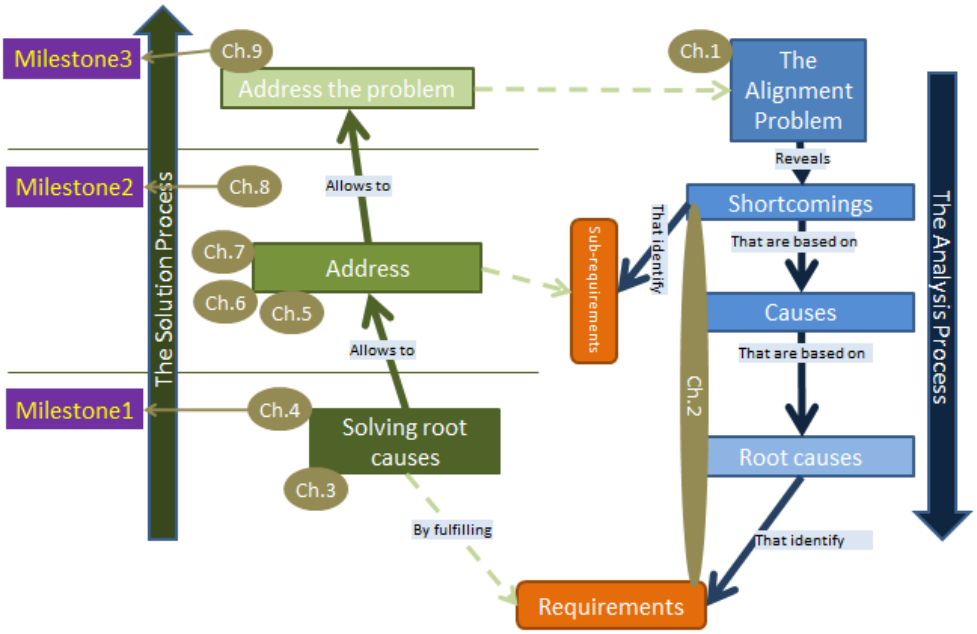

Figure 1.8 Reading map.

Chapter 2 State of the art

2.1 Context

This chapter performs the steps from 2 till 6 in the methodology explained in 1.5. The first section is dedicated for step 2: *understand the process*. This section identifies the approaches to align the UI development and the SE development activities. The interest of the alignment is to address usability in the developed product. Therefore, approaches that integrate usability in SDLCs are in the scope of interest to be identified. However, the identification of the approaches is based on a visit to the history to understand the processes followed to address the human factors crisis since the time it was announced, which is traced in the evolution of the HCI domain.

The sections that follow analyze each approach according to the methodology as:

- State limitations in the approach as depicted in the literature.
- Gather data on each approach in the form of conclusion statements. These **conclusions are steps in the reasoning towards the root cause** behind the limitations in the approach.
- Reason on the approach to reveal the root cause.

The last section is dedicated to elicit requirements to solve the root cause. The elicited requirements delimit the space of possible solutions. The last section implements the step 6 in the methodology.

2.2 Identifying usability integration approaches

2.2.1 A visit to the history

Carroll provides a reading of the history of the HCI from the point of view of the evolution of HCI models, theories and frameworks [Carroll2003]. He reports that HCI was originally a joining of software engineering and human factors engineering. He denotes the following evolution steps of HCI towards a multi-disciplinary domain:

- In the 70s: the software engineering crisis due to the use of the slow and unreliable waterfall development process. **Software Human Factors also faced crisis: it was positioned at the end of the waterfall. Thus, it was involved after main design decisions had been taken. It was limited to cosmetic differences in software products.**
- Towards the end of 70s till mid-80s: Cognitive Science had coalesced as a multi-disciplinary project **encompassing linguistics, anthropology, philosophy, psychology, and computer science** (figure 2.1). According to Carroll, one principle of Cognitive Science is the representational theory of mind. A second principle was that an effective multidisciplinary science should be capable of supporting and benefiting from application to real problems, and HCI was that field. The initial vision of HCI consisted in bringing Cognitive Science methods and theories to provide substantive guidance at the early stages of software development.
- In the mid-1980s: HCI saw itself as an emerging scientific discipline. New scientific ideas entered the HCI mainstream. Some sources of new ideas are:
  - Growing multidisciplinary constituency of cognitive science itself: Social psychologists, anthropologists, and sociologists entered the cognitive-science discourse, sometimes taking HCI as their empirical touchstone.
  - Increasing internationalization of HCI: New conferences were held in Europe, application of activity theory[3] like in the work of Bannon and Bodker [Bannon1991].
  - Technology: personal computers, word-processing and spreadsheets. Later in the 90s: The web and networking, technological support for graphics and visualization, audio and video. Handheld computers and cellular phones.
- In the mid-1990s: HCI encompassed nearly all of social and behavioral science. The tremendous range of empirical methods

[3] Activity theory was originally developed in what is known now as Russia.

and scientific concepts in routine use in HCI has been a source of strength as the field grew to address new problems and issues encompassing new technologies and new applications.

Carroll's reading helps to understand the evolution of HCI as a multi-disciplinary domain. To focus more on the UI domain and how it evolved, we review a different reading to the history from the UI and modeling point of view.

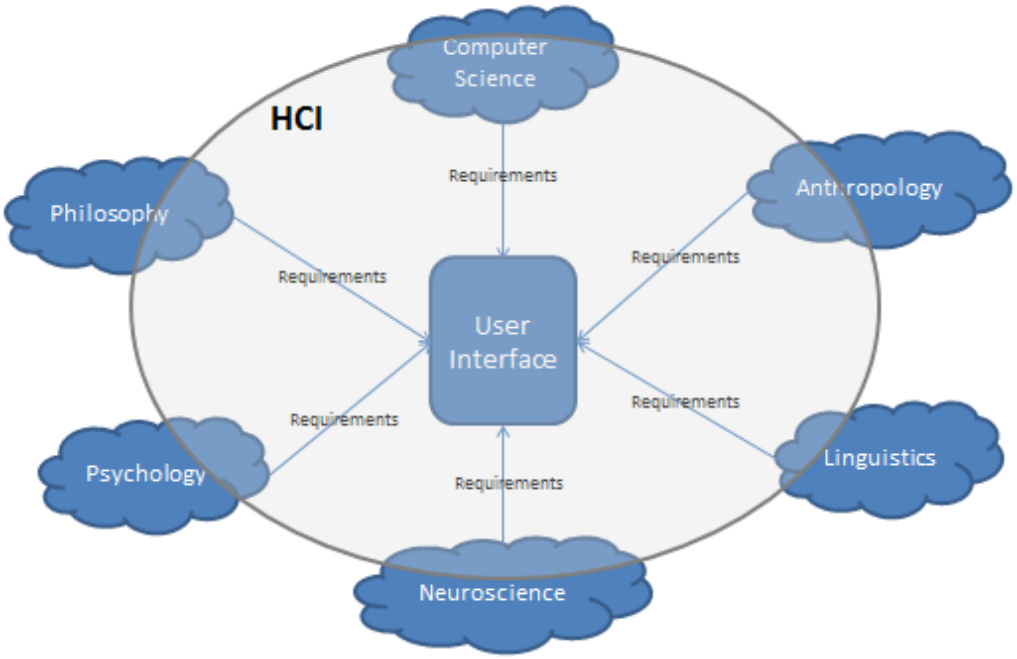

Figure 2.1 HCI is emerged from different disciplines/fields to support the UI development.

In their reading of the history of MB-UIs (Model-Based User-Interface) [Meixner2011], Meixner, Paternò and Vanderdonckt trace evolution steps towards current UI models. We present interesting steps from their historical selection in the context of this thesis.

- In the mid-1980s, the HCI community focused on supporting GUI developers by creating **tools and frameworks**.

- In the beginning of 90s, the community focused more on **modeling approaches**: abstract relevant aspects of a UI, and the search for one universal UI model.
- From 1995-2000: the HCI community focused on extending the UI model by integrating other distinct models expressing high-level semantics of a UI: **task** and dialog models.
- From 2000-2004: support new interaction platforms and devices, smartphones or PDAs.
- From 2005-now: **context sensitive** UIs for different platforms, devices and modalities. The Cameleon Reference Framework (CRF) [Calvary2002] and the UsiXML [UsiXML2007] are two examples characterizing this period.

Figure 2.2 illustrates the evolution of HCI and MB-UI on the same time axis. Carroll's reading provides an understanding of how HCI evolved and why we needed to extend UI models in the mid-1990s. In the 1990s and before, UI models addressed the **technical part**[4] of the domain. The coalescence of cognitive science into the HCI domain provided an opportunity to MB-UI approaches to fill the analysis gap. Researchers profited from this opportunity by extending the UI by mainly **the task model**. This can be traced back in the literature to the work of Myers [Myers1995] who suggested more investigation in the area of **high-level UI models**. Even a decade before, Green [1985] suggested **starting the UI development from a task model**.

The history of MB-UI shows also that researchers are satisfied with the task-based approach, since **the focus drifted from solving human factors crisis into addressing new challenges** like new interaction platforms, devices, context-sensitive and different modalities. But did the task-based approach really solve the crisis? We will address this question in depth in our analysis.

This visit to the history depicts the following findings:

1- The task model is used to fulfill the need for a high level model to extend the UI domain.
2- The focus of the HCI community is drifted from solving the

[4] The part of the UI that is covered in UI development tools and frameworks is the technical part.

human factors crisis to addressing new challenges like: new interaction platforms, devices, context sensitive and different modalities.

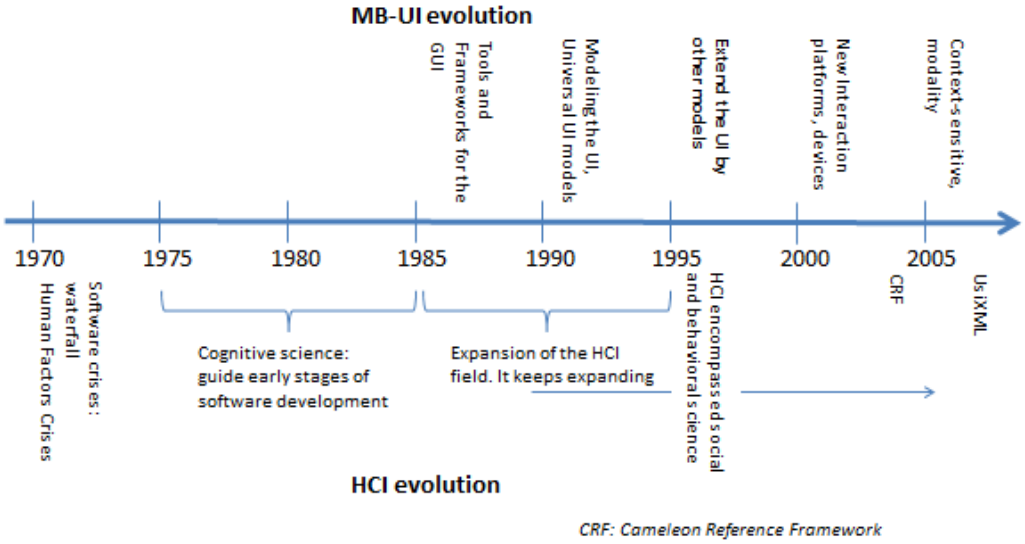

Figure 2.2 the historical evolution of MB-UI and the HCI

2.2.2 The Identified approaches

Knowledge on usability is accumulating in the HCI domain in the form of usability guidelines. Software Engineering has the processes to develop the software product. From a broad view, integrating usability in the software development is a question of integrating the usability knowledge in the software development process. This integration can basically follow one of two approaches: manual or model-based.

The analysis of usability knowledge (usability guidelines) is important to understand the process of accumulating and organizing these them. Understanding how software engineering development lifecycles address the human crisis to develop usable products is also important.

Manual integration of usability in the software development is followed in software development tools and frameworks. This approach is the oldest and the dominant approach in the 80s and 90s. It is still widely used nowadays. This approach is important to analyse and fits in the focus of this thesis.

Model-Based approaches that gained focus during the 90s are also in the focus of interest of this thesis. These approaches include models of usability knowledge in addition to models for the UI and the context of use.

Although the task-based approach is part of model-based approaches, it deserves a separate analysis. This is because it emerged in the 90s to fulfill the need to use a high-level model for the UI.

Our visit to the history identified 5 approaches to analyse in order to identify shortcomings in each approach and identify the root cause behind these shortcomings. The identified approaches are depicted in figure 2.3. These five approaches are:

1- Usability Guidelines: Analyse how usability knowledge is gathered and organized in the HCI domain. The analysis focuses on the knowledge itself, not how to use it.
2- UI development frameworks and tools: This is the oldest approach to develop the UI. It depicts the manual approach to the UI development. This approach is the mostly used approach among others nowadays in the industry.
3- Modelling approaches: Employs models in the development of the UI. These models include modelling usability knowledge, context of use and the UI domain in addition to developing methods and tools to generate the UI from these models.
4- Task-based approach: Although this approach is part of the modelling approaches, but it is of a special interest because the task model is considered a high-level model to start the UI development with. It deserves to be discussed independently.
5- Software engineering approaches: understand how software engineering addresses the human factors crisis in their processes in order to develop a usable software.

With usability development approaches identified, we identify in the following sections shortcomings in each approach and reason on the root cause behind these limitations. Once the root cause is identified, we elicit requirements to solve it.

The identified root cause is not meant to be the only root cause for limitations in an approach. **Other root causes may exist as well**. Overall, identifying and solving a root cause is of a great interest as it can highly impact limitations in the approach (a solution with high leverage points) when compared to low impact resulting from superficial solutions (solu-

tions with low leverage points) by addressing limitations directly.

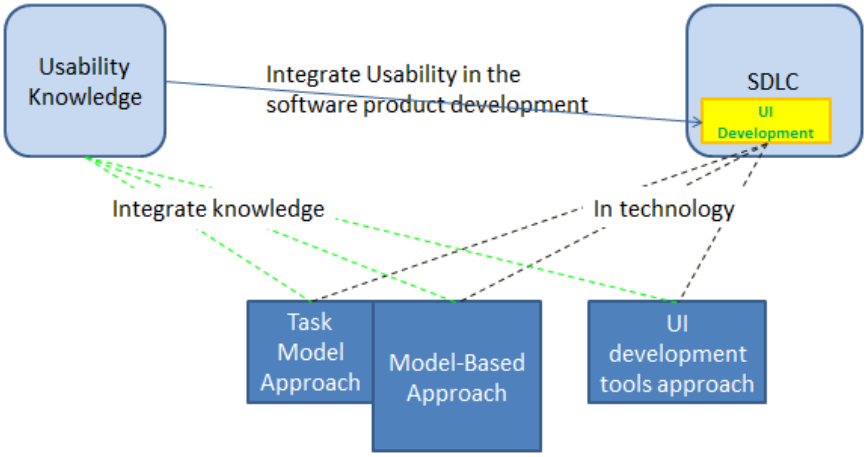

Figure 2.3 Relationship between analyzed approaches.

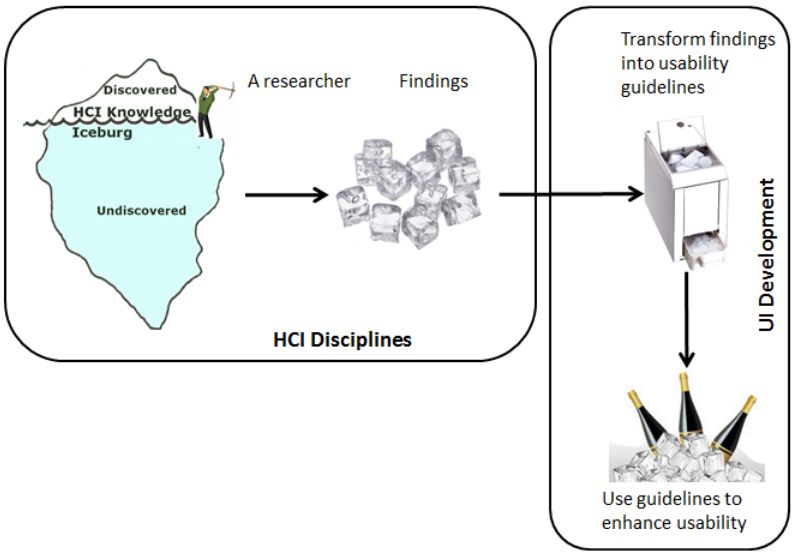

Figure 2.4 The general process to employ usability guidelines in the UI development.

2.3 Usability guidelines

2.3.1 Description

This approach is **primarily**[5] based on research conducted in Cognitive Science. It aims at identifying usability knowledge, organizing and formulating them into guidelines. The promise is to provide substantial resources to identify usability requirements since the early phases of the development, thus enhancing the usability and the acceptance of end-users for the software product. The general process[6] to gather usability guidelines is depicted in the figure 2.4.

The interest in this approach is based on the common sense of the problem of integrating usability in the software UI: human factors need to be addressed at the early stages of the development process. The solution is to identify these factors *a priori*. The promise of the approach is explained in figure 2.5. The figure depicts how usability guidelines may fill the human factors knowledge gap to start the project at the right level, and consequently integrate usability in the software product.

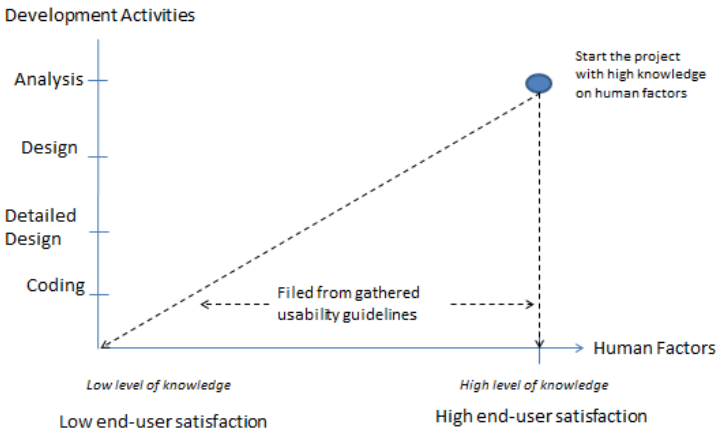

Figure 2.5 The promise of the usability guidelines approach to integrate usability in the product development.

[5] There are also design guidelines that are coming from the UI design and validation phases.

[6] The general process covers design guidelines as Computer Science is part of the HCI. See figure 2.1.

Much research was carried following this approach. A huge body of knowledge in the domain exists nowadays. The reader may refer to the book of Vanderdonckt [Vanderdonckt2007] for a list of 3700 usability guidelines. There are also domain-specific sets of guidelines, like cultural-specific usability guidelines [Rau2011] or web-specific guidelines [Mariage2005a]. Besides, guidelines are sometimes gathered based on a specific cognitive model, like the list of guidelines based on Hofstede's culture model in [Callahan2005].

To give the reader an idea of these guidelines, we enlist some of them below:

- General usability guidelines:
    - G1: Cursor movement should be minimal.
    - G2: Error messages should provide a specific feedback.
    - G3: The system should provide needed help.
    - G4: Colors should be distinguishable
- Cultural guidelines for designers towards an international UI design:
    -G5: Allow extra space for text.
    - G6: For menu design, provide orientation compatible with the language being presented.
    - G7: People classify things differently according to their cultural backgrounds (a UI navigation-related issue).

2.3.2 Use of Guidelines

If we want to use the guidelines in the UI development, we need to understand their use across different development phases by different development roles. Vanderdonckt [Vanderdonckt1999] explained the use of guidelines across different phases as:

1- Specification Phase: a set of guidelines is delimited as requirements for the future UI.
2- The Design Phase: guidelines are exploited in order to decide an appropriate value for each design option by considering the context.

3- Prototyping Phase: guidelines are exploited to quickly obtain a static or working UI prototype that can be showed, tested and evaluated.
4- Programming Phase: guidelines are gathered to guide, orient, decide, ensure a UI development within the developing environment for the targeted computing platform;
5- Evaluation Phase: the resulting UI is evaluated with respect to guidelines that are often selected in previous phases of the design process;
6- Documentation and Certification Phase: guidelines which have been manipulated in previous phases are employed for documenting an interactive application.

2.3.3 Limitations

The approach is consistent with traditional engineering development approaches, which enforce a complete set of requirements at the end of the analysis phase. Even though, there are limitations on the approach that are reported by researchers in the literature. We identify two types of limitations: (1) limitations concerning the knowledge themselves and (2) limitations concerning integration in the UI development (the use of guidelines). Table 2.1 below represents these limitations with the reference to them. We comment on each type in the following.

1- Limitations on the knowledge in the guidelines

The main limitation of the knowledge in usability guidelines is that knowledge is **huge and scattered**. Besides, **incompleteness** of usability knowledge is intrinsic. Therefore, it is hard to assume that the approach can highly satisfy its promise: *provide the resources needed to elicit usability requirements a priori*.

Furthermore, we should note that these limitations are **primarily** related to cognitive science fields, not to the UI domain. The knowledge in these domains is not complete. For instance, we do not have yet a complete model on the user in psychology nor in other fields.

In some domains, even the **validity** of findings is doubted. The reader may review some critics to culture studies in [Henrich2010]. Based on a

statistical study of psychology and culture literatures, Henrich states that most of conducted experiments are applied on western university students. Extrapolating these findings on different people is not that simple. With these critics, concerns on the validity of usability guidelines are magnified.

Id	Type	Limitation	Reference
1	Knowledge	Huge number of guidelines, which is increasing with time.	[Vanderdonckt1999]
2	Knowledge	Guidelines variation in validity.	
3	Integration	Guidelines variation in level of details.	
4	Integration	Guidelines variation in relevant development phase of use.	
5	Integration	Guidelines variation in target development role.	
6	Integration	Insufficient guidelines classification.	
7	Knowledge	The expressiveness and the trust in the guideline validity heavily depend on the guideline source.	[Mariage2005b]
8	Integration	Nearly all guidelines require some interpretation	
9	Knowledge	The jargon used in the initial guideline may slow down designers.	
10	Integration	Usability guidelines can be sorted by linguistic level.	
11	Integration	Applying and checking guidelines require varying workloads	

Table 2.1 Limitations of the usability guidelines approach.

These limitations should be addressed in the main field of interest before importing resulting guidelines to the UI development**. Addressing such limitations is out of the scope of this thesis**.

2- Limitations on the integration of usability guidelines in the UI development

The limitations 3 and 8 depict an important property of guidelines: they differ in the **level of abstraction**. For instance, some usability guidelines may recommend stylistic modifications on the UI. See for example the guideline G5 mentioned before. Other guidelines identify higher-level aspects to consider, like the guideline G7. Low-level of abstraction guidelines are interesting to the UI developer role because they are concrete enough to be implemented. High-level of abstraction guidelines might be interesting to the interaction designer role because they might stimulate new ways of thinking while designing the interaction [Khaddam2014].

The limitations 4 and 5 depict the importance of considering development phases and roles when integrating usability guidelines: At what phase of the development a guidelines will be used? And who will use it? These limitations motivate the need for a role-based classification of guidelines, because such a classification can provide answers to these questions. The lack of classifications of usability guides is depicted in the limitation 6.

From the above, classification of guidelines plays an intrinsic role in the integration of guidelines in the software development. Firstly, there is a lack of classification of usability guidelines. A classification that supports integration of usability guidelines in the development should:

1- Depict different levels of details.
2- A role-based classification of usability guidelines: map guidelines to a development step/phase and/or to a development role.

Furthermore, the limitation 10 enlightens the possibility to classify usability guidelines linguistically, which is a kind of classification on different levels of details. Lower linguistic levels would contain concrete and tangible knowledge, while high-level linguistic levels contain abstract knowledge. Limitation 11 depicts that a linguistic classification might minimize workload when applying and checking usability guidelines.

The linguistic classification is promising because it provides different levels of details. However, for the linguistic classification to be used in the integration of usability, it needs to be mapped to development roles and/or UI development activities.

In the above, we illustrated the importance of the classification and its relation to other integration limitations. However, the HCI community spent a lot of time working on usability guidelines and classification systems. It is quite interesting to review existing classifications for usability guidelines, looking for a classification that supports integration of usability guidelines in the development. Having a role-based classification that classifies usability knowledge on different levels of details of usability guidelines would address all integration limitations enlisted in table 2.1.

2.3.4 Classification of guidelines

Many classifications of usability guidelines exist, each represents a specific point of view: by ergonomic rules [Streveler1990], by usability factor, as in ISO 9241 standard [ISO9241], by interaction style, as in Mayhew's guide [Mayhew1992], by widget, as in IBM CUA style guide [IBM1993], according to an object-oriented model on input/output, as in ITHACA report [Scapin1989], by importance level, as in Banks's standard [Banks1983], or by type of widget, as in Farenc's ERGOVAL automatic evaluation tool [Farenc1997].

All of the above classifications do not provide an answer to the important questions to the integration of usability guidelines in the development: What guidelines to consider during a development phase? And who will do that? Therefore, none of them can be considered as a role-based classification of usability guidelines.

It is hard to claim that there is no role-based classification of usability guidelines in the literature. But if one existed, it would not be that deeply hidden as it is promising to the integration of usability guidelines in the development. The absence of role-based classification should have a good reason behind this issue. In the next section, we try to figure out why the literature does not have a role-based classification of usability guidelines.

2.3.5 Analyzing limitations on the integration

The HCI community spent a lot of time and effort working on the classification of usability guidelines. What are the challenges to define a role-based classification? Why is it hard to establish such a classification, although it is promising to the integration of usability guidelines in the development?

Assume we could establish a role-based classification, where the main categories are development roles. A usability guideline that is classified at category "*C1*" shall be integrated by the development role "*C1*". The question that arises is: which SDLC one can follow to integrate usability guidelines in the development? Apparently, SDLCs that do not define the same/equivalent roles in the classification are excluded from the list of choices. Therefore, a role-based classification of usability guidelines limits their applicability to compatible SDLCs. Compatibility here means the SDLC must define the same/equivalent roles denoted in the classification categories.

In practice, development roles are flexible and vary from one SDLC to another. Even when the same role name is used, it may not be assigned the same activities. Furthermore, the definition of activities depends on the perspective to the UI as we have seen in sections 1.2.1 and 1.2.2. Note that a task analyst role exists in the HCI perspective but not in the SE perspective.

In a conclusion, a role-based classification for usability guidelines limits their applicability to compatible SDLCs. The classification of guidelines is a tedious and time-consuming work to perform. Such an effort can be justified if the results are used in various SDLCs.

2.3.6 Towards an alternative classification

If role-based classification is not a good idea as it looks from the first sight, what alternative can we have?

At the core of each usability guideline, there are issues related to the UI concepts that designate what concepts on the interface are affected by this guideline. For instance, the guideline "Colors should be distinguishable" impacts the UI concept "the color". The guideline "Allow extra space for text" impacts the placement of widgets on the screen. The guideline "People classify things differently according to their cultural backgrounds" impacts navigation concepts (neither placement nor colors concepts). Therefore, identifying the affected UI concepts in a guideline would designate its impact on the UI, and consequently identify the activities to perform to integrate the guideline in the development. Table 2.2 depicts such a classification with concepts and activities impacted by the guideline.

Guideline	UI concept	UI Activity
Colors should be distinguishable	Color	Modify color
Allow extra space for text	Placement of widgets on the screen	Place widgets on the screen
People classify things differently according to their cultural backgrounds.	Navigation	Design the navigation on the UI

Table 2.2 A classification of guidelines based on the impacted UI concept.

Such a classification would classify usability guidelines based on **the nature of change embedded in the guideline**. Note that some guidelines might affect more than one UI concept, which might indicate the need for more refinement of such a guideline.

Although the proposed classification in table 2.2 is NOT a role-based classification, but it designates the activities needed to integrate a guideline in the development. These activities can be mapped to a development role when a SDLC is selected and adopted for a project. We explain this in the coming example.

Assume we have the classification of guidelines in table 2.2. Assume also that an SDLC is already chosen to a specific project. The chosen SDLC defines development roles in its own way, and defines what activities are performed by each role. The first step to integrate the usability guidelines in table 2.2 is to identify concerned role(s) with each guideline. This can be easily achieved based on the identified activities (third column in table 2.2). We need to ask simple questions like "What role is responsible for modifying colors on the UI/placing widgets on the screen/designing navigation?" The answer to these questions leads to mapping guidelines to roles as defined in the SDLC. Possible answers are (for illustrative purposes only):

- Case 1: The SDLC defines only one role to consider usability in the development role: The UI designer role. In this case, all the above guidelines are assigned to the UI designer role.
- Case 2: The SDLC defines the roles of Usability Expert and UI Designer the first and second guideline in table 2.2 can be as-

signed to the UI developer while the third can be assigned to the Usability Expert role. We assume here that the navigation design is the responsibility of the Usability Expert, not the UI Designer.

The process to integrate classified guidelines, based on their impact on UI concepts, takes the form:

(1) identify roles in the SDLC;
(2) identify activities carried out by each role;
(3) map classified guidelines to these activities.

This alternative classification of usability guidelines implicitly implies a classification of UI concepts (column 2 in table 2.2). Once we have a classification of UI concepts, we can establish such an alternative classification. Therefore, the classification we need is a classification of UI concepts for guidelines to be classified based on their impact on UI concepts.

In order for the classification of UI concepts to be used in classifying usability guidelines, it should fulfill the following requirements (with justification explained inline):

1) Have a countable and reasonable number of categories: if the number of categories is large[7], the classification becomes useless.
2) Reflect the level of abstraction of the UI concept: This would enable abstract guidelines to be mapped to UI concepts on the same level of abstraction.
3) Classify all UI concepts: any UI concept should belong to a category in this classification.
4) Univocally classify UI concepts: any UI concept belongs to one and only one category in this classification. This is important to avoid ambiguity when classifying a guideline that impacts this UI concept.
5) Conceptually independent from each other: a UI concept can be manipulated by a single UI activity. If two activities can manipulate the same concept, a guideline may be assigned to two different activities and consequently to two different roles.

[7] Large number of classification categories increases the mental load of the person using the classification. The rule of 5 to7 categories might apply here.

2.3.7 Summary

The approach is very successful in integrating UI development in the development life cycle, it provides valuable assets to understand requirements. It proofed also fruitful to the UI design like in internationalization support, where concrete rules on differences among cultures are used to develop adaptable algorithms and components to cultures (like the bidi[8] algorithm to write text in a bi-directional way: from left to right).

We identify two sets of limitations in the approach. The first set concerns the nature of these guidelines and cannot be addressed in this thesis: they are scattered, huge, incomplete, and sometimes its validity is questioned. The cause behind this set of limitations is related to their fields of origin in cognitive science.

The second set of limitations concerns their use in the UI development, where classification plays an important role. We argued that a role-based classification enhances their integration in the UI development, but limits their applicability in different SDLCs.

We argue that an alternative classification, based on classification of their impact on the UI, enables their integration independently of SDLCs. To establish such a classification, we argued that we should find first a classification for UI concepts that meets the requirements stated before.

2.4 The task model approach

The promise of this approach is to employ a task model to analyze interaction requirements for a specific project. As we have seen in section 2.2.1, this approach evolved because limitations in UI models at the design phase were identified; therefore we need high-level models to start with. The task model provided a solution for this issue.

The task model approach looks like a realistic refinement to the holistic approach (adopted in the usability guidelines approach): identify all human factor issues *a priori* based on cognitive science knowledge. The task model's approach is: it is possible to identify human factors per project.

[8] More on the bidi algorithm can be found at this url: https://en.wikipedia.org/wiki/Bi-directional_text

This promise of the task model is depicted in figure 2.6.

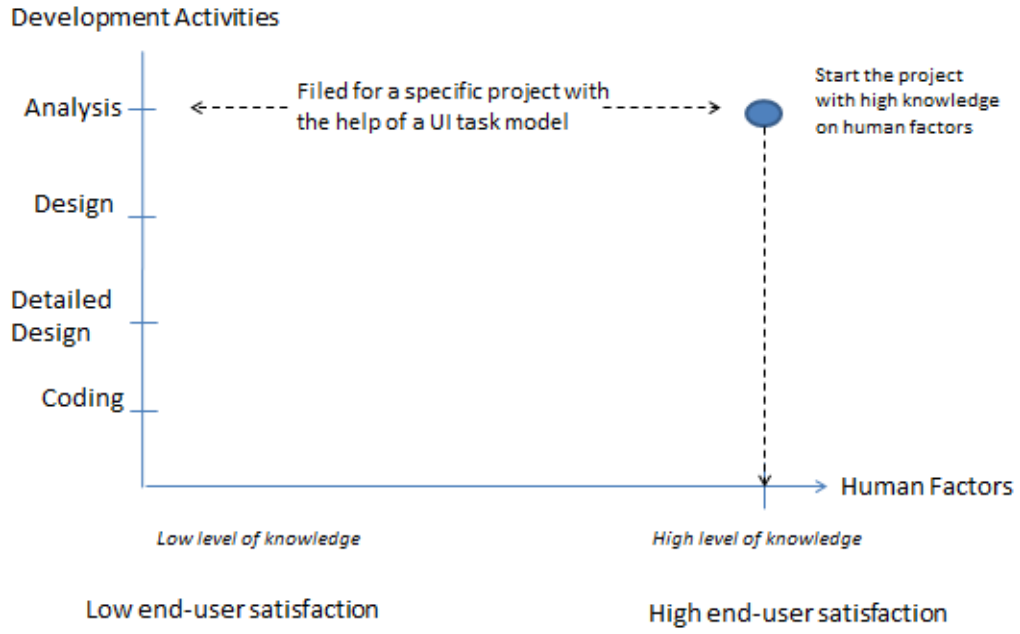

Figure 2.6 The promise of task models approach to integrate in the product development.

2.4.1 Task models and UI task models

In the late 1960s, Annett and Duncan offered a means to describe system in terms of goals and sub-goals, with feedback loops in nested hierarchies [Annett1967]. This developed later into the specification of HTA (Hierarchical Task Analysis). The theory is based on a goal-directed behavior where we identify sub-goals in a hierarchy linked by plans. Plans describe how to perform a sub-goal and determine conditions to trigger a sub-goal. The performance towards a goal can be achieved at multiple levels of analysis.

The purpose of a task model is to describe the world and how the work is performed [Diaper2004]. Diaper considers that there are two things at the core of any task analytic model:

1- A description of the world.

2- An account of how work is performed in the described world.

HTA is the central approach to existing task models [Stanton2006]. While HTA reasons on describing the current world and how it is envisioned in the future with enhanced performance, task models used in the UI development refine this reasoning by describing how the work will be performed using an interactive computational device. We call the latter models **UI task models**.

We consider UI task models as integration models[9]. They integrate cognitive science knowledge on how users perform tasks in the real world, into the UI. An example of UI task models is CTT [Paterno1997]. An example of non-UI task models is HTA.

Based on the above, we conclude:

Conclusion Task_1: UI task models are integration models. They integrate Cognitive Science with the UI domain.

2.4.2 From UI task models to the UI

Let's have a look at how UI task models integrate cognitive science knowledge in the UI domain.

1- The "task" concept and the UI

The task model has mainly one concept: the "task". **This concept belongs to the real world, not the UI domain**. The UI does not have the concept "task", although it is possible for some readers to get confused by this statement. We support it by:

1- No UI language or development framework has the notion of a task (Java, Android, .Net, HTML among others).
2- If the task concept belongs to the UI domain, we should expect that reverse engineering from the UI to the task model to be easy. This is because all we need is to identify tasks in a UI and then establish the task model. Well, this is not the case. Limbourg reported difficulties in reverse engineering from AUI to Task and domain models in the Cameleon Reference Framework

[9] When we analyze Modeling Approaches, the term "integration models" will be generalized to other UI models.

(CRF) [Calvary2002] due to the important conceptual gap between these two levels [Limbourg2009].

Based on the above, we conclude:

Conclusion Task_2: The concept "Task" is not a UI concept

Based on the conclusion Task_2, we can reach another conclusion:

Conclusion Task_3: Translation of the concept "Task" to a UI concept(s) is required[10]

2- Translation of the concept "task" into the UI

UI task models translate the concept "task" into a concept (or a set of them) in the technical UI domain (a widget, a set of widgets, a screen, a full UI among others). But where is this translation happening and what are the consequences of such a translation?

Surprisingly, **the translation is embedded in the analysis phase**, not after the completed task model. This is related to the way tasks are decomposed into sub-tasks and when to stop this decomposition. To explain this point, we discuss three representative examples on UI task models: CTT, K-MAD [Caffiau2010] and a layered task approach [Pribeanu2006], seeking to identify what criteria are employed to stop task decomposition. We call these criteria: Task Decomposition Stopping Criteria (**TDSC**).

Identification of tasks in CTT is based on identification of activities in the scenario. Tasks in the hierarchy can be added to represent a semantic grouping of identified activity tasks. Anyway, the decomposition may continue and **stop at the granularity of identifying needed user input element**.

K-MAD is a hierarchical model of tasks from the most general one (root) to the most detailed ones: **elementary actions**.

The layered approach to UI task modeling aims at developing the task

[10] Some HCI researchers may argue that software engineering should integrate the «task» concept into the software engineering domain in order to integrate task models in the software development. We think that as long as HCI cannot provide more than prototypes from task models, it is still early for such a suggestion.

model on several levels. Pribeanu proposes a layered task model on 3 levels: a functional, a planning and an operational layer [Pribeanu2006]. The distinction between these levels is based on different TDSCs:

1- Functional: tasks associated with the same business goal. This criterion applies for task decomposition at **functional level** for the mapping of application functions to user tasks.
2- Semantic: task performing an operation onto the same domain object. This criterion is applied to separate tasks which refer to the same object or to the same operation (add new, delete) or to the same interaction method (when several methods are available to accomplish a goal). It is relevant for **both functional and operational level** and helps in the identification of unit tasks.
3- Task object: tasks performing operations with the same interaction object or external object. The criterion is relevant for the **operational level** and helps in the identification of basic tasks.
4- User and work: tasks are performed by the same user (playing a given role) and are denoting a similar work (manual, interactive, communication). The criterion is mainly relevant for **cooperative tasks**.
5- Temporal: tasks denoting specific temporal constraints (like repetitive or optional performance). The criterion is relevant for the **representation of temporal constraints among tasks**.

Although all the above criteria are interesting in general, the criteria that are pertinent for the context of this thesis are those stop the decomposition at the operational layer, the lowest in the hierarchy. This is because the UI will be generated from this layer.

The decomposition of tasks at the operational task level stops at the **basic task**. The basic task is the task that is using a single interaction object or a single external object or serves a communicational goal. There are two types of basic tasks, according to the interaction object type: information control and function control. Each type of basic tasks is mapped to **an Abstract Interaction Object (AIO) when generating the UI from the task model**.

All the TDSCs above illustrated an implicit translation of the concept "task" from the real world into the UI domain by mapping leaf tasks to input elements, elementary actions or even interaction objects (concrete or abstract ones). The UI task modeler needs to "**Think UI**": **perceive**

the design of the UI when decomposing tasks.

Many examples can be found in the literature on task models to demonstrate the "Think UI" during the task decomposition. We choose to present one from the first books on UI task models using CTT, by Paterno [Paterno2000, pp.77,78]. The book gives an example of a UI and the accompanying task model to demonstrate the layout of presentation of an agenda UI. Figure 2.7 is reproduced from that book as is.

The task "EnterDate" enforces the use of a specific interaction style: use three widgets (text boxes to fill the date). But interaction styles are design aspects, and the choice should be given to the designer and not enforced by the task analysis. This example demonstrates that the task analyst needs to think UI during the task analysis.

Of course, the same UI can be produced differently like in figure 2.8 but using a different interaction style (one text box for the date instead of three). In this example, if we apply the task stopping criterion on the granularity of needed user input elements, then the task analysis should have stopped at the task "EnterDate".

But one may argue as: what is the problem of employing a TDSC based on the UI design? The ultimate goal is to build the UI, and the task analysis is refined until it reaches the UI design space using such a stopping criterion.

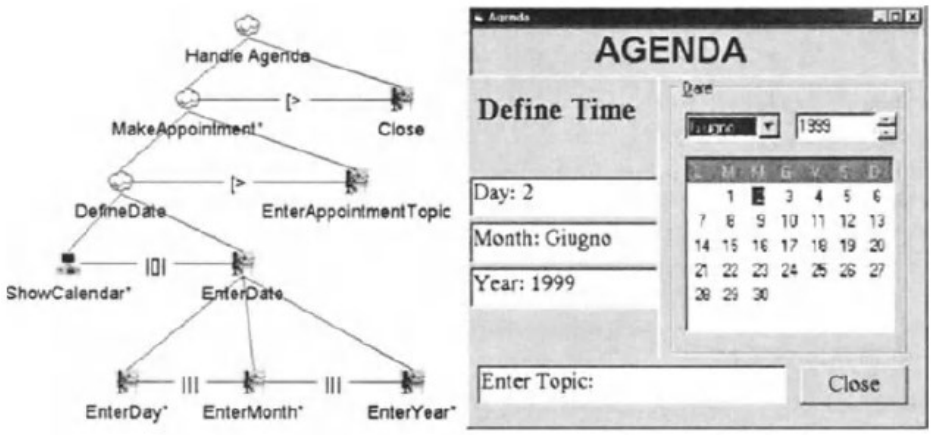

Figure 2.7 An example of using task model and the accompanying UI

At a first glance, a task stopping criterion based on user input element

(concrete or abstract one) or finite user actions might seem sound. One can accept that performing a task on a UI requires an input element. But in fact, **not every input element** on the UI is created to **perform a task**. Some input elements are created for other purposes, like enabling navigation between screens. Figure 2.9 illustrates an example of an input element that performs a task (the "send" button), and other two inputs (maximize and minimize) that perform no task. Here we decide that "send" is a task because we can perceive the goal of the user as: "the user's goal is to send the email". But on the other side, it is hard to accept that maximizing or minimizing the screen is a goal of the user (at least in this context).

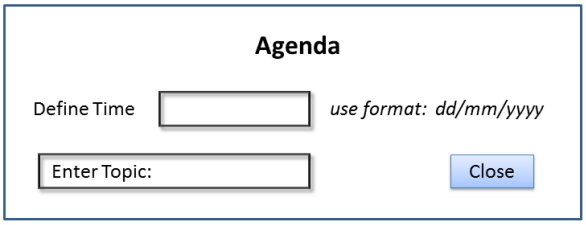

Figure 2.8 An alternative design for UI

Conclusion Task_4: UI task decomposition stopping criterion is coming from the design phase

Anyway, the coupling between UI task models and the technical UI is quite interesting. Each of them solves a problem of the other. The task model has the problem of defining the decomposition stopping criteria (see next section), which is solved by the UI. The UI has the problem of addressing human factors, which is handled by the task model.

1- On task decomposition stopping criteria

TDSC is one of the critical aspects in HTA. The TDSC in HTA is determined through the probability of failure (P) multiplied by the cost of failure (C) [Annett1971]. When the estimation of this formula P*C is acceptable, the analyst can stop the decomposition. Although this formula is simple enough, its applicability is hard.

The P*C criterion is different from those used in UI task models. P and C are related to the human's world and performance of tasks. The main difference relies in being independent from the UI domain (the real world boundaries are respected in this criterion).

GOMS, yet another UI task model, is established after the complete HTA analysis. Thus it also inherits the problem of defining the decomposition stopping criteria. GOMS could not make use of the TDSC as defined in HTA. It employs a pragmatic criterion that is based on judgment calls [Kieras2004]. The analyst needs to decide when to stop decomposition relying on a psychological theory or model for how people do the work. Although GOMS TDSC is not related to the UI domain, but judgment calls are far from being considered as an objective criteria.

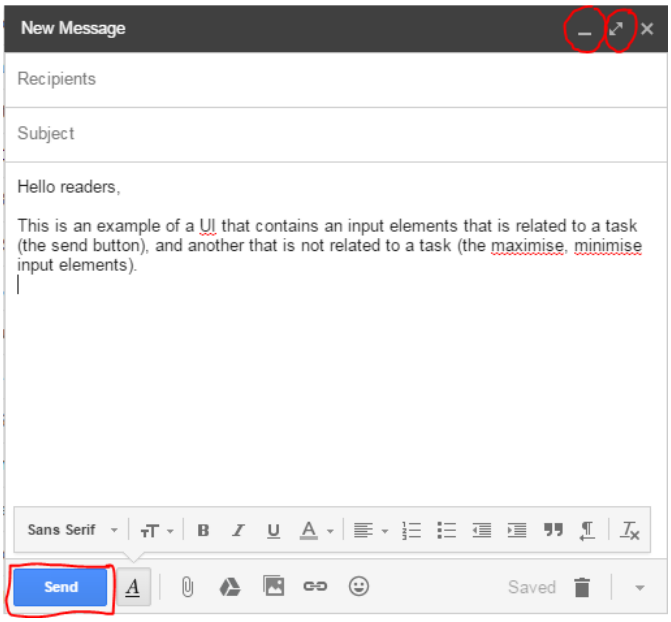

Figure 2.9 An example of input elements, where some do not carry out a task

We argued and identified a set of conclusions on task models. Based on these conclusions, we can analyze limitations facing this approach and understand the causes of these limitations. We do this in the coming sections.

2.4.3 Limitations

What limitations and problems are reported on UI task models? Our literature review on task models did not reveal in-depth critics or unsatisfactions of the task models approach. What one can probably find are soft critics that motivate enhancements from one task model to another. It looks like researchers tend not to criticize UI task models.

The author of this thesis participated in a workshop on task models in IHM'15[11] in Toulouse. Participants from different universities from France discussed their experience in teaching the task model in universities. Many interesting discussions revealed the feeling among participants on difficulties of convincing students to use the task model in their projects. A one-hour brain storming session on how to enhance the teaching experience of UI task models, and attract students to use them, took place at the end of the workshop. Organizers of the workshop had a published work-in-progress paper based on three experiments on students and teaching experience [Oliveira2015]. These experiments reveal a main limitation of the approach: *students do not use the task model unless forced to*. Realizing that some of those students will become future designers, this limitation risks in many cases to be transferred to the industry as well. The workshop revealed that the HCI community, although shy to criticize the task model, senses a problem in the adoption of UI task models.

Furthermore, task models are part of Model-Based approaches. Researchers reported on the low penetration rate of research models into the industry. Hutchinson assessed the use of MDE in industry [Hutch2011]. Modeling languages used by participants of a questionnaire showed only one participant who is using UsiXML [UsiXML2007] in contrast to more than 80 participants using the Class Diagram model. UsiXML employs a task model among other models, which should be valued by the industry as it provides a methodological guidance since the early phase of the project, but reality follows a different logic than expected.

Motti *et al* evaluated the use of model-based approaches by different practitioners working for Information Technology companies, with different expertise and roles [Motti2013]. Based on a user-survey, they report on the adoption of models that almost half of the participants do not use models (16 out of 30). 6 participants use MDE, 11 participants use UML diagrams among which 3 use MDE. The study does not tell how many out of the 6 participants who use MDE are using the task model. In the best case, we count 6 participants out of 30 who might be using the task model, which still represents a low penetration rate in the industry.

[11] http://ihm2015.afihm.org/

2.4.4 Intermediate cause analysis

A UI task-model evangelist may describe the problem as a utility/usability issue: We do not have good and usable tools for the students. The fallacious assumption here is: if we have appropriate tools for the task model, the industry will use them. This assumption is in fact based on the implicit assumption: the task model has no problem and is appropriate for the UI development.

There is a clearly illustrated methodological problem by the conclusion 4: *UI task decomposition stopping criterion is coming from the design phase*. The methodological problem is related to the role given to the UI task model in the UI development process: is the UI task model intended to be used at the analysis phase, or both the analysis and design phases?

Conclusion Task_5: UI Task models have a methodological problem

This problem can be stated also as a problem of separation of concerns in the task model: what is on the analysis phase and what is on the design phase in the software development phases. This problem is more serious than having appropriate tools for UI task models. It may better explain why people do not use UI task models in the software industry. This methodological problem causes problems in real development of a UI as well, because UI developers do not know what should be modeled in the task model and what should be added apart at later phases of the development. Besides, it is not clear how to integrate the result of the UI task model in the software development: is the resulting UI a prototype? Or part of the final UI? In both cases, how the UI developer will move on to develop the final UI?

2.4.5 Options to solve the methodological problem in UI task models

How to solve the UI task model methodological problem? The clear answer is to give a well-defined role for the UI task model: to cover the **analysis** phase, the **design** phase or **both**. Let's analyze each option.

1- The UI task model for the analysis only

If we intend to use the UI task model on the analysis phase only (to fill the analysis gap, as originally requested by the community in the 90s. See

section 2.2.1), we certainly need to look for a TDSC that is independent from the design phase and consequently independently from the UI, because the UI is at the design phase.

2- The UI task model for the analysis and the design

If we accept to extend the role of the UI task model to cover the design space, we need to increase the expressiveness of the UI task model. This might be achieved by importing concepts from the UI domain (the design).

K-MAD and HAMSTERS are examples where the task model is extended to cover the design phase, by introducing new concepts like ports and mapping with the data model (design phase concepts).

If we accept such an extension to the role of the UI task model, we should be careful to the consequences. Such an extension comes with the tradeoff of increasing complexity of the task model by added concepts. Efforts to enhance expressiveness of the task model result in increased complexity. See figure 2.10 from [Limbourg2006]. This figure draws the conclusion that the task model cannot model the design space in a simple way.

Expressiveness is a main concern when using task models to cover the design space. This is because if the expressiveness of a model is less than development tools, we cannot expect more than producing prototypes from these models.

However, even if we accept to extend the role of the task model to partially cover the design, we need to pay attention to consistency between the task model and other models employed in the design (if several models are used). This is the case of CRF, where the task model is used at the highest level. Below the task model are Abstract User Interface (AUI) model; that cover the modality-independent design space, and the Concrete User Interface (CUI) model; that covers the modality-dependent design space. Consistency is a concern in CRF because we need to ensure that modification on one of these models is consistent with upper models. For example: modifying the AUI (like splitting a container into two) does not affect interaction as specified in the upper task model.

In a conclusion on the options to solve the methodological problem in task models: depending on the role given to the UI task model, the modeler is confronted with different challenges. Figure 2.11 depicts

these challenges.

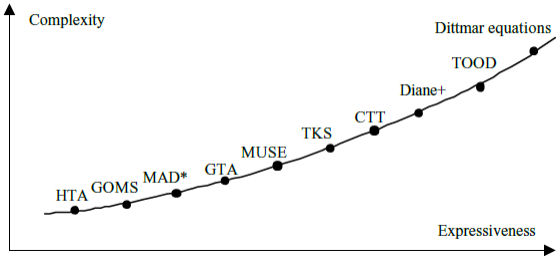

Figure 2.10 Expressiveness versus complexity of task models [Limbourg2006][12]

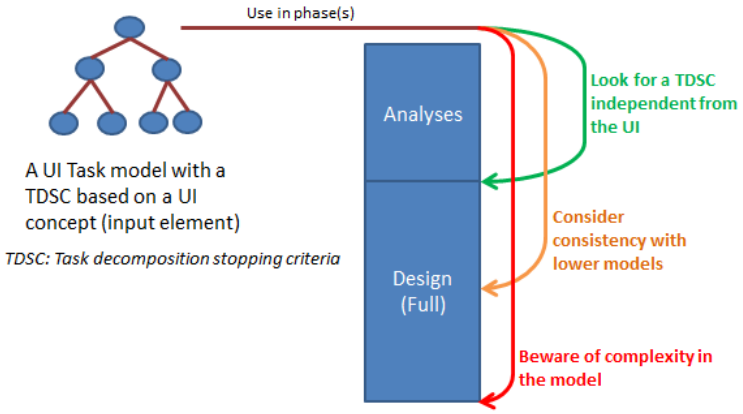

Figure 2.11 Challenges to UI task modelers based on type of task decomposition stopping criteria and the role given to the UI task model.

2.4.6 The root cause of limitations of UI task models approach

Extending the role of the task model to cover the design fully or partially (the first two options discussed previously) may not be that fruitful to

[12] GTA : GroupWare Task Analysis [VanDerVeer1999], TKS : Task Knowledge Structure [Johnson1989], MUSE: Method for Usability Engineering [Lim1994], Diane+: [Barthet1996], TOOD : Task Object-Oriented Description [Mahfoudhi2001], Dittmar: [Dittmar2000].

the UI development. This is because the task model was exploited originally to solve the analysis gap in addressing human factors, not design problems[13]. Such approaches would look like moving the problem forward, which makes resulting solutions superficial.

The approach to use the task model on the analysis phase only is more consistent with the historical motivation behind using the task model. However, we are facing one big challenge: find a TDSC that is independent from the UI (independent from the design space). We need something like the P*C criterion used in HTA but for UI task models.

While it is possible to find a UI-independent TDSC[14], a closer look at the issue reveals that it will not completely solve all the issues. A task model that stops at the analysis phase would look no more than a formal description of the analysis document (a scenario, a user-story or others). This is because it will contain only the upper level tasks of currently known UI task models (leave tasks related to the design are filtered by the UI-independent TDSC). Consequently, even with a UI-independent TDSC, a translation of the UI task model to the UI model is needed (conclusion task_3).

The challenge with the translation is to avoid producing any UI concept that belongs to the design space. If the translation produces UI concepts, it would be covering the design space partially or fully, which is the op-

[13] We disagree with approaches that use the task model to generate a UI prototype. Recall that in our visit to the history in section 2.2.1, we illustrated that the task model was needed to start the UI design from a high-level model that fills the analysis gap. Approaches that use the task model to produce prototypes are putting the task model in an unfair comparison with prototyping tools. Prototyping tools are design tools and more powerful than task models because they are more expressive: prototyping tools use the same concepts that the UI designer and the user are used to (unlike the task concept in the task model). Besides, prototyping tools are more intuitive at the design space than the task model, because the prototype can directly express what is in the mind of the designer, which is not the case with task models. However, prototypes are unable to cover the analysis phase, where the task model excels and provides a unique value that is not provided by any other tool. The task model can elicit interaction requirements, while prototype tools learn on these requirements by trial and repetition.

[14] We will get back to this issue in chapter 5 when presenting our proposed task model

tion we tried to avoid at the beginning of this section. Our only choice is to **translate the "task" concept into an analysis UI concept**: a UI concept that belongs to the analysis phase and not the design space.

Although the above requirement looks strange enough, but it leads to another strange question: **How the UI should look at the analysis phase?** This question looks non-sense from the common perspective to the UI, that perceives it at the design phase and not before. But this question reveals the root cause of the problem that prevents establishing a task model on the analysis phase only:

The current perspective to the UI does not allow perceiving the UI at the analysis phase

Thus, it is a perspective problem, not a modeling one!

The current perspective to the UI places all UI concepts at the design phase (input widgets, output widgets, placement of elements, colors, and so on). It considers all input elements on a UI at the same level of abstraction[15][16]. Therefore, translating the task concept will always result in covering the design space partially or fully. Consequently, it is impossible to employ the task model in the UI development without having an impact on the UI design and limiting the designer options. This is the root cause of the "***Think UI***".

We mentioned before in this section that an input element might represent a task, but not every input element on the UI represents a task. An **input element** that is identified by a **task** should **not** be at **the same level of abstraction** as an **input element** that defines **navigation,** because the task input element is identified at the analysis level (the same level where the task is identified), while the navigation is identified based on a design decision (like the need to adapt to a graphical screen dimensions). A submit button on a form (like a submit button to pay an invoice) is not at the same level of abstraction like a "next" or "previous"

[15] Level of abstraction here is: the design level.

[16] UI models may abstract UI concepts introducing Abstract Interaction Objects (AIOs) vs Concrete ones, others may abstract them like: Platform-Dependent and Platform-Dependent. These abstractions are all in the design phase. No abstraction promotes UI concepts to the analysis phase. The task concept is the only concept that was believed to belong to the analysis phase.

buttons in a wizard-like UI. Our current perspective to the UI depicts them all at the same level of abstraction.

Assume we have a perspective that allows defining UI elements at different levels of abstractions[17]. The UI task model, with a UI-independent TDSC can be translated into analysis-UI elements (UI elements defined at the analysis level of abstraction). This means, the designer can continue to design the UI using design-UI elements (UI elements at the design level of abstraction). Such separation of UI concepts would provide the required separation of concerns to solve the methodological problem of UI task models. It opens wide all the design options to the designer, while respecting requirements enforced by the analysis-only task model and the resulting analysis-UI.

2.4.7 Conclusion

The integration of the UI task model in the UI development is not successful because it has a methodological problem: the role given to the task model is not clear if it covers the analysis and/or the design phases. This is a problem in the task decomposition stopping criterion that is based on UI concepts. A solution to this methodological problem is to find a stopping criterion that is independent from the UI, and thus limits the role of the task model to the analysis phase only.

Our in-depth analysis revealed that the cause behind the methodological problem relies in the current perspective to the UI development that forces the UI task model to cover the design space partially or fully.

If we want to employ the task model to cover the UI analysis phase only, we need a new perspective that allows defining UI concepts on several levels of abstractions: analysis-UI, design-UI and so on. With such a perspective, the task model can translate the task concept into analysis-UI concepts, thus leaving the design space open for the designer to manipulate design-UI concepts.

The result of the analysis is depicted in figure 2.12.

[17] Levels of abstractions here depict abstraction of UI concepts from the analysis phase till the lowest phase, not only abstraction at the design phase. Refer to the previous footnote.

2.5 Modeling Approaches

A **model** is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system [Bezivin2001]. It is a simplification of reality by considering important aspects, of the domain[18], to the problem in hands, in order to reason about it. **Simplicity** allows focusing on these aspects to understand and then address the problem.

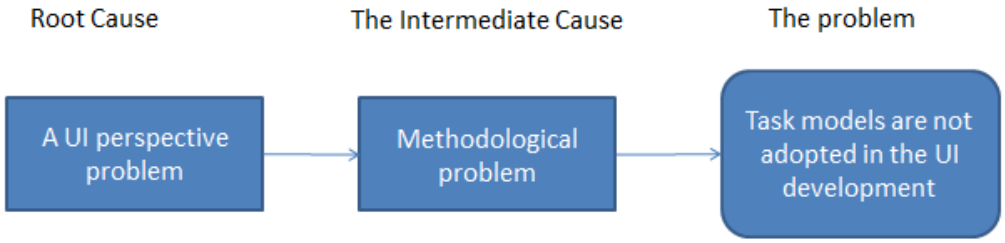

Figure 2.12 The result of the task approach analysis with the root cause of the approach integrating problem identified.

Model-Based User Interface (MB-UI) development approaches gained a lot of attention from the HCI community due to their potential benefits during the past decades: when the model changes, reasoning activities remain unchanged and are straightforwardly applied. Such assets are mainly inherited from model-based development and Model-Driven Engineering (MDE). Technical advantages claimed by MDE proponents are improvements in productivity, portability, maintainability and interoperability [Kleppe2003]. Another potential benefit is the support of exploring alternative designs, which allow the production of different designs for different contexts of use [Masson2010] while maintaining consistency [Pilemalm2012].

MB-UI modelling addresses the challenge of UI development mainly by:

1) Abstraction: to simplify the domain by considering important aspects to the problem.
2) Engineering discipline: to systematically develop the UI.

[18] The term «domain» is used in its general meaning. It is not the term used in Model-Driven Engineering.

In the coming, we investigate how modeling[19] is used in the UI development and how it is used to incorporate HCI knowledge into the UI development following a model-based approach.

2.5.1 The general process to use models in the UI development

We trace modelling approaches on the general process to employ usability guidelines in the UI development, as depicted in figure 2.4 and redrawn in figure 2.13. Firstly, modelling is used in different fields constituting HCI. Examples are culture models, psychological models of the user and the behaviour, among others. These models represent findings in these fields with important aspects to reasoning. These models can be seen as a formal representation of knowledge in these fields. Anyway, they are **not** the models used in the UI development. We call them: **HCI-Fields Models**

The second type of models is concerned with transforming HCI fields' findings into HCI-related models. This is also a kind of formalism of important aspects on usability and consequently the UI. We call these models: **HCI integration models**. These models are derived from HCI fields' models and encapsulate knowledge on usability. An example is: a user model for HCI. This user model might be different from that in the psychology field. It is derived from the psychology user model and focuses on important aspects to the UI.

HCI integration models are used to bridge the technical space to generate the final UI. They can use **technical UI models** as an intermediate model towards final UIs. The technical UI domain is the model that contains solely concepts that appear in UI development frameworks and tools. Technical UI models are models in the technical UI domain. Examples are XML UI models like AUI and CUI in CRF, XUL[20], XAML[21], among others. Using technical UI models allows for more flexibility to manually tweak the resulting UI.

The **model-based approach** in this section **targets** mainly **HCI inte-**

[19] Our investigation is not limited to MDE which defines the triad: Model, Method and Tool. We investigate Model-Based approaches, where MDE is a sub-category.

[20] https://developer.mozilla.org/En/XUL

[21] http://en.wikipedia.org/wiki/List_ of_user_interface_markup_languages

gration models. We seek to understand their promise, limitations as well as the causes behind these limitations. Below, we summarise and define the three different types of models we discussed above.

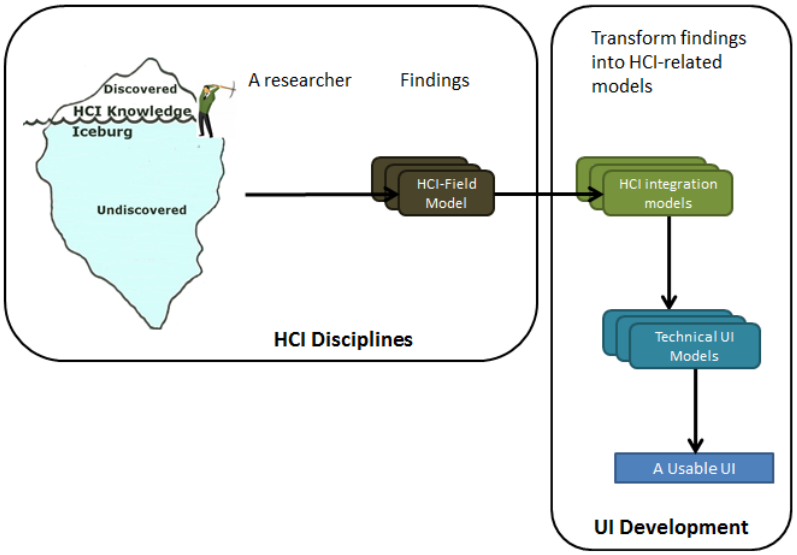

Figure 2.13 The generic process to employ usability knowledge in the UI development in a model-based approach.

Technical UI: the UI domain and concepts that appear in UI development frameworks and tools. Technical[22] UI models target this domain.

HCI-Fields Models: these are models from HCI fields. Examples are: Hofstede's model for culture, user models, context of use models among others.

HCI Integration models: these are models that aim to address usability in UIs. They import concepts from HCI-fields (non-UI concepts) to serve in the development of a usable UI. CRF and UsiXML fit in this type.

[22] The word "technical" is coming from the evolution history of the UI. The UI was considered a technical module in the software and addressed using development tools. Modeling approaches that address the UI as a module in the software development abstract the technical aspects of the UI, therefore, we call them technical UI models.

2.5.2 The promise of HCI integration models

The model-based approach to integrate HCI knowledge in the UI development might be described as: with models of usability knowledge concerning the user and the surrounding (the context of use), a usable UI can be built. The UI is manipulated through high-level models: modify the high-level model and the UI is adapted accordingly. These models are non-technical; therefore, a non-technician (like a usability expert) can manipulate them and consequently build the appropriate UI.

The promise of this approach is to (1) *foster the UI development* because models can be built and modified easily by non-technicians, (2) *encapsulate usability knowledge complexity into a model*. The promise is depicted in figure 2.14. This figure illustrates how modeling reduces the analysis gap and shortens the development process.

2.5.3 An example on HCI integration models

An example is the well-known Cameleon Reference Framework (CRF) and its instantiation language UsiXML [UsiXML2007].

UsiXML employs several models to develop the UI. Some of these models model the technical UI domain. These models are: AUI, CUI and FUI. UsiXML also employs a task model to fill the analysis gap. The task model is an integration model because it contains non-UI concepts (the "task" concept). There are also different models for the context of use: platform, user and environment. Context models are also integration models because they define non-UI concepts.

Integration models **bridge the knowledge** from HCI fields into the UI. This is the case of the user model in UsiXML. The user model contains the cognitive science knowledge on the user, which is pertinent to the UI. Another example is employing a culture's model to generate the appropriate UI. The culture model becomes an integration model of culture knowledge and the UI.

2.5.4 Limitations of integration models

Several HCI integration models limitations are reported by the community. Let's review part of them.

Firstly, we do not see high penetration of research models into the industry. Hutchinson assessed the use of MDE in industry [Hutch2011]. Modeling languages used by participants of a questionnaire showed only one participant who is using UsiXML [UsiXML2007] in contrast to more than 80 participants using the Class Diagram model. **The industry tends to use UI development frameworks more than research UI models**. A similar conclusion is reached also by Motti *et al* in [Motti2013].

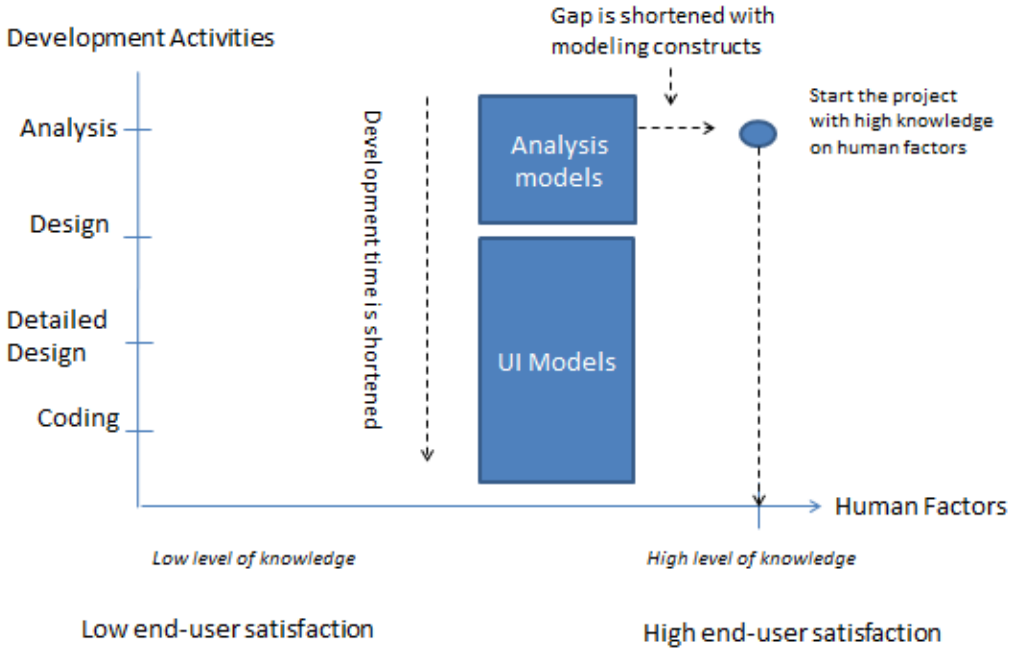

Figure 2.14 The promise of the modeling approach to integrate usability in the UI development.

On the other hand, Vanderdonckt described limitations of UI modelling approaches as: "**wide walls** (different types of UI) and **high ceilings** (capabilities)" [Vanderdonckt2008], which denotes limitations on the types of UIs generated and their capabilities. He also reported **traceability** concerns and **increased complexity** in transformation engines.

Coutaz and Calvary in [Coutaz2012] report on limitations of MDE in plasticity of the UI, in the light of their experience in the CRF. **Transformations** are hard to express, and **separation** between **design time and run time** has drawbacks, like models run out of sync with running

code. They also criticize the Concrete User Interface (CUI) model for "not only lags behind innovation, but bridles creativity".

In the coming sections, we try to reveal the causes behind these limitations. For this purpose, we analyze all three types of models mentioned above because they all play part in the model-based approach to employ usability knowledge in the UI development.

2.5.5 Analyzing different modeling approaches

In this section, we state some conclusions on each type of modelling.

1- HCI Fields Models

Fields constituting the HCI discipline have a huge body of knowledge associated. It is impossible to have one comprehensive model for a field. Models focus on a reasonable number of aspects. Therefore, it is expected to establish several models of the human that differ in aspects covered depending on what kind of reasoning is expected, like description, comparison among others.

The more reasoning activities need to be covered, the more expressive the model should be. Therefore, employing one model for all possible reasonings becomes complicated, if possible.

For example, several models for the culture exist. They might differ in covered aspects. Hofstede's model [Hofstede1997] covers a different set of culture dimensions than other models like in [Hall1989]. Most of the time, it is hard to transform one model for a culture into another. Thus, even a unified model for a specific reasoning (reasoning in this example is "a description") is not possible.

Conclusion HCI_MOD_1: A unified model for a specific reasoning is not possible.

If creating a universal model for a discipline in HCI is practically impossible, then it is the case when thinking about a model for all HCI fields together.

Conclusion HCI_MOD_2: Employing one model for all possible reasonings is not possible

Models should avoid complexity by minimizing the number of covered concepts. When the number of concepts covered is huge, the modeling

approach loses interest (simplicity). Thus, **it is intrinsically inevitable to have several models, even for the same reasoning.**

Conclusion HCI_MOD_3: it is intrinsically inevitable to have several models, even for the same reasoning

Another characteristic of modeling in HCI fields is that these fields are open, in the sense that they are not completely explored and continue to evolve by new discoveries. This makes models subject to later evolutions, as they might be modified by newly discovered knowledge that might affect the meta-model behind.

Conclusion HCI_MOD_4: HCI models are subject to evolutions

2- Technical UI Models

Different models for the technical UI are created and used by the industry. Many XML models exist in the industry that model UI concepts, like Java Swing, XAML from Microsoft and Android UI language from Google. Developers practice these models in products, which assume a certain degree of satisfaction.

The technical UI is **simple**, compared to HCI fields' models. It is a specification in the machine. It contains a countable number of concepts, basically input elements and output elements. Therefore, a technical UI model can be complete and stable. Stability here means the modification and evolution of these models is not probable.

Conclusion TECH_MOD_1: Technical UI models can be stable and complete

This conclusion may explain why technical UI models have advantages in the UI development. These advantages include covering the widest range of the design space, and providing the highest flexibility in the UI development. Simply said: because they are complete and stable models.

On the other side, modeling the technical UI independently carries some limitations. In section 2.2 we noted the problem faced this modelling approach in the history: missing of high-level models. Therefore, they cannot be used to fulfil the analysis gap, and require other analysis models. For this reason, we attribute them as **methodologically weak models**: they need to be extended with high-level models to establish a com-

plete development step-wise methodology.

Conclusion TECH_MOD_2: Technical UI models are methodologically weak.

Technical UI models and frameworks are the concern of the forth approach and we will get back to them later. For the moment, the two aforementioned conclusions are enough for the reader to follow our analysis of the approach in this section.

3- HCI Integration Models

As we mentioned before, HCI integration models are derived from HCI fields' knowledge and models. They abstract relevant usability concepts from these fields. Concepts in integration models primarily belong to HCI fields. Therefore, we can state the first conclusion:

Conclusion INTEG_MOD_1: HCI Integration models are derived from HCI models, with focus on important aspects to the UI.

Based on conclusions INTEG_MOD_1 and HCI_MOD_4, we can conclude another:

Conclusion INTEG_MOD_2: HCI Integration models are subject to evolutions.

From a broad view, generating a UI from integration models is a matter of translating concepts in these models into technical UI concepts. This is usually embedded in transformation engines.

Translating integration models' concepts into technical UI concepts can be a complex task to perform manually. For instance, the cultural background of the user designates a set of adaptations on the UI: language, writing direction among other preferences. All these adaptations can be encapsulated in a culture model for the user. Based on the user's culture, we can carry out a set of adaptations automatically. The translation complexity is embedded in the transformation engine allowing a smooth and seamless translation of culture concepts into technical UI ones. Eventually, the translation of concepts is vital because the machine understands technical UI concepts only.

By introducing translations between concepts, we are coupling the integration model with the technical UI model. This is because the transla-

tion requires subtle information on the UI architecture. For example: how are widgets modelled? How is navigation modelled? What are the existing architectural modules/components and for what purposes? Without answers to these questions, translation cannot be performed. Without a translation, the integration model cannot be employed in the UI development.

Assume we need to change the reading direction on a widget (a usability knowledge encapsulated in a culture integration model), we need to know how the widget is modelled (a technical UI domain) and how to change its reading direction. **Thus, for an HCI integration model to be fruitful in the UI development, it needs to be coupled with a technical UI model**.

The translation always depends on the destination model. In MDE, the transformation engine depends on the source model and the target model. The concept of a transformation engine that is independent of the destination does not exist. Therefore, we state the conclusion:

Conclusion INTEG_MOD_3: To use an HCI integration model in the UI development, it requires coupling with a technical UI model.

When coupling the HCI integration model with the technical UI model, we are creating a new model to develop the UI. This model is neither a technical UI model nor a HCI field one. It is a new model that contains concepts of both disciplines. This UI model has the following limitations:

1- Although the mapping between the knowledge level and the technical UI is simplified by embedding the translation in the transformation engine, it introduces a limitation in the types of UIs it can produce. This is because we can embed a **limited set of translations** only. If we miss a possible translation, we have no alternative than manually tweaking the resulting technical UI.
2- The new UI model is subject to evolution because the HCI integration model is an evolving model (INTEG_MOD_2). Thus, **we are introducing un-stability[23] to the UI development**

[23] The model is no more stable. It will evolve with the HCI integration model.

model. Compare this to the stable technical UI model (TECH_MOD_1).

3- The resulting model is **more complex** than both models. It contains more concepts, and the modeler needs to understand all of them. Compare this to the simpler technical UI model.

The coupling problem looks responsible for limitations of the approach. Limitations on produced types of UIs and capabilities are related to the coupling. Increased complexity in transformation engines is mainly due to effort to embed more translations. These efforts are not promising because models are unstable and the number of translations cannot be foreseen or fixed. The more expressive we try to make the model, by introducing new concepts, the more complex it gets. To address the approach limitations effectively, we need to focus our analysis on the coupling as a cause behind these limitations

2.5.6 To model or not to model: The UI modeling Dilemma

Based on the analysis we made before, the coupling problem a property of integration models. Therefore, we can avoid the coupling problem if we avoid using integration models. The question that arises is: Do we really need to use integration models?

If we model the technical UI we can create complete and stable UI models but with a methodological problem in considering usability requirements at the analysis phase (TECH_MOD_2). If we extend the technical UI with concepts from HCI fields we can methodologically address usability requirements, but with incomplete models that consequently generate limited UIs.

The UI modelling dilemma is: technical UI models are complete[24] and stable but methodologically weak. Extending technical UI models with concepts from HCI fields solves the methodology problem but creates

[24] Note that completeness of a model is according to the domain it addresses. Technical UI models address the technical UI domain, which is a closed domain. Please refer to conclusions previously stated in the Modeling Approaches section.

incomplete[25] models that are subject to evolution.

Existing modelling approaches address the UI dilemma by favouring the methodology over the completeness of the model. Thus, they employ cognitive models to fill the analysis gap. Although such an approach guides the UI development, it imposes constraints on UI design options through transformations from the integration model to the UI (the translation). The final UI is controlled with capabilities of the upper model: the integration model. When the UI development starts with an incomplete model, it ends with low design coverage.

The industry prefers stable and complete models. Therefore, they solve the dilemma by adopting technical UI models.

2.5.7 Towards solving the UI dilemma

Is there a way to address this dilemma? Modelling has always been a good tool to answer questions. Modeling this dilemma may help to find a solution.

A simplified model of the dilemma is depicted in figure 2.15. This figure depicts how HCI integration models are exploited in modelling approaches to fill the analysis gap. A translation of HCI fields' concepts is used to generate technical UI concepts. The resulting hybrid model is depicted in the right part.

Figure 2.15 illustrates why coupling is complex: it consists of a **translation of high-level concepts into lower level ones across two different domains**. Red triangles in the figure depict abstract concepts from HCI fields, while blue circles depict concrete technical UI concepts.

Examples of abstract HCI concepts are: a user, a task, culture, an environment, etc. Examples of concrete technical UI concepts are: a button, a container, a widget, a window, a mouse event, a hand gesture, etc.

A translation from one level of abstraction to another is possible and may not impose limitations. It may take the form of a **refinement** of the abstract concept into a more concrete one. On the other side, translating

[25] Incompleteness here because the domain addressed by the extended model has changed from the technical UI domain (a closed domain) to the HCI domain (an open domain).

a concept into another domain is certainly more complex and limiting. This is because several possible interpretations exist. This complexity is increased if we translate an abstract concept into a concrete concept in another domain.

CRF gives a good example of translation from high-level concepts to more concrete ones in the same domain. AUI, CUI and FUI models are at different levels of abstraction of the technical UI domain. Translation (refinement) of concepts from a high-level to a lower one is straight forward. AUI has abstract UI concepts. Translation into CUI refines AUI concepts to become platform-independent concepts. These are refined later to become concrete platform-dependent concepts at the FUI level. Translation is in the same domain and is predictable from one level to another.

However, this is not the case when translating from the task model to the AUI. Different interpretations can be given on how to create AUI containers from the task model [Tran2012, Betri2004], and none of them can possibly cover all design-space possibilities. Translation from the task model to the AUI is an example of a translation from a high-level concept in a domain (the concept task in the domain of task analysis) to a lower-level concept in another domain (AUI containers at the design level of the technical UI domain).

To ease the complexity of this coupling, we may decompose the translation into two steps: refine and translate. The process would become as:

Option 1:

1- Refine concepts in the same domain.
2- Translate across domains.

Option 2:

1- Translate across domains.
2- Refine concepts in the other domain.

This first option requires prolonging the HCI integration model to cover the **lower phases of the development**. This way, we can translate/refine abstract HCI concepts into more concrete ones in the same domain. When the development is completed, we can translate these concrete HCI concepts into UI ones at the same level of abstraction.

This is depicted in figure 2.16.

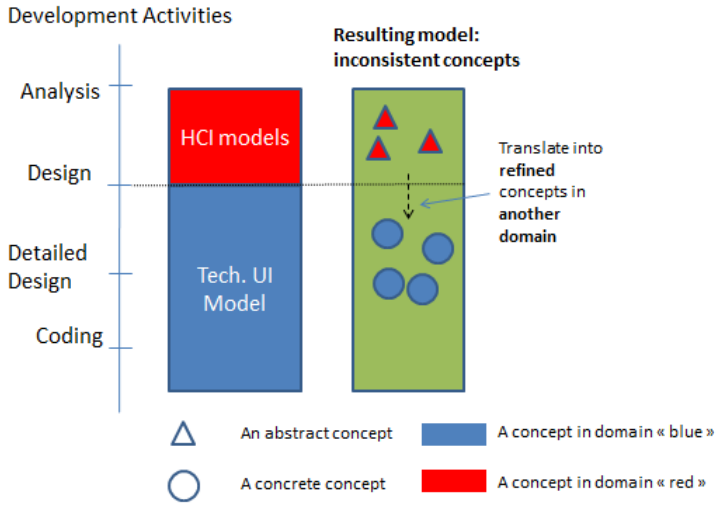

Figure 2.15 Modelling the UI dilemma

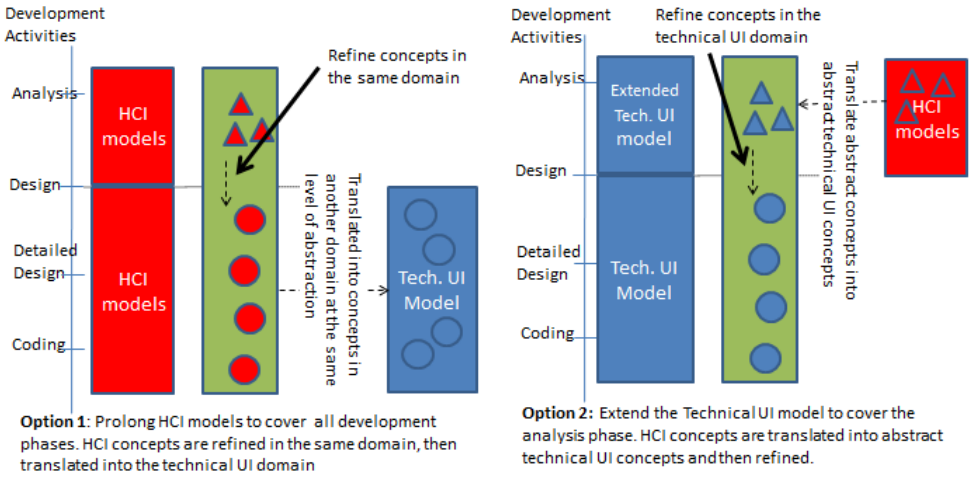

Figure 2.16 Options to address the UI dilemma: Extend HCI models or extend the technical UI model.

However, such an option looks far from being realistic. Prolonging HCI fields to cover development phases is beyond the interest of researchers in these fields. A researcher in psychology is interested in the human behaviour, and not with the development of the UI for this user. This ex-

cludes the first option.

The second option suggests **extending the technical UI domain to cover the analysis phase with concepts from the technical UI domain** itself. Abstract HCI concepts could be translated into abstract technical UI ones, which in turn can be refined into concrete UI concepts. This way, a concept like "task" or "user" does not appear in the UI domain. It would appear only in the HCI integration model and will be translated into a UI concept at the analysis phase.

If the second option were feasible, it would provide a **solution to the dilemma** and not an enhancement by reducing complexity of translation. In fact, extending the technical UI at the analysis phase would negate the conclusion TECH_MOD_2 (Technical UI models are methodologically weak). The technical UI would no more need HCI models to fill the analysis phase. It would be able fill the analysis phase with technical-UI analysis concepts. Therefore, the extended technical UI model can employ a methodology where HCI models are supporters and not key players.

The second option suggests to abstract technical UI concepts at the analysis phase (back to strange suggestions). Therefore, we would expect to have an analysis-UI and a design-UI. This requires a new perspective to the UI development that allows perceiving the UI at the analysis phase and to be refined at the design phase and lower. As long as such a perspective is not developed, the UI modeling dilemma persists. Thus, the root cause of the UI dilemma, and consequently limitations in the modeling approach is the current perspective to the UI development that makes the technical UI methodologically weak.

The current perspective to the UI is the cause of the UI dilemma

Thus, the UI dilemma is a perspective problem, not a modeling one!

The current perspective to the UI places all UI concepts at the design phase. It considers input elements on a UI at the same level of abstraction. That is why we need to import concepts from HCI fields into the UI model to fill the analysis gap.

Perceiving the UI at the analysis phase does not mean we do not need HCI fields anymore. It simply re-defines their role: they can guide the UI at the analysis phase. For instance, a task model, a user model, a culture

model or any other HCI model, can be used at the analysis phase to gather interaction requirements of the end-user. These models will not impact the UI at the design phase. They will impact UI concepts at the analysis phase, leaving the design space to the designer.

2.5.8 Conclusion

The current perspective to develop the UI places the technical UI at the design phase. This makes technical UI models methodologically weak. This methodological problem requires HCI fields' models to fill the analysis gap and therefore enable a methodological enactment of the UI development, while considering usability.

The integration of HCI fields in the UI development causes the UI dilemma. This dilemma is responsible for limitations in the approach. This dilemma cannot be solved from the current perspective of the UI, which considers the UI as a weak domain and cannot be perceived at the analysis phase.

To better integrate HCI models and the UI, one should search for a different perspective to the UI. We need a perspective that allows perceiving the UI at the analysis phase. This would allow a more flexible integration of HCI integration models in the UI by enabling translations at the same level of abstraction. Besides, this would minimize restrictions on the UI design, because the integration affects the analysis phase only. The design is a refinement of the analysis, just like the relation between the analysis and the design in the software development.

The result of the analysis is depicted in figure 2.17.

2.6 UI development tools approach

2.6.1 Description

Many UI development tools exist. Almost every general programming language provides tools to develop the UI. Examples include Java with JavaSwing library, .Net from Microsoft with XAML, Android from Google with a dedicated UI framework. The web technology unified the UI through the use of HTML as common UI language to be interpreted by web browsers.

These tools and development frameworks employ models in their

frameworks. The main difference in these models from modelling approaches relies in the targeted user of these models. While UI modeling approaches in general tend to eliminate the developer role (from design to execution), technical UI models target the developer.

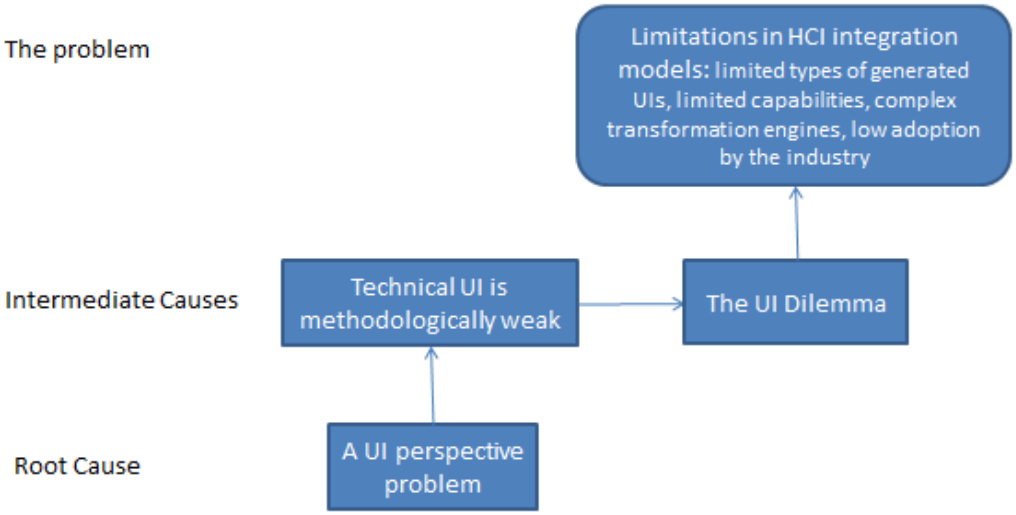

Figure 2.17 The result of the modeling approach analysis with the root cause of the approach limitations.

The objective of UI development tools is to provide the developer with the technical ability to develop the widest range of UI designs, supporting new interaction techniques and styles. They also address other qualities like: minimize effort to develop the UI (like using What You See Is What You Get editors), enhance interoperability (like using XML for representation) among others. From this point, the **promise of this approach** to integrate usability in the UI development is by leaving the "what" to the developer and supporting the "how": The developer decides what needs to be done, and the development tool makes it simple. Their approach is to minimize development time and thus faster contact with the user and low modification effort. Therefore, minimize the severity of usability issues on the project by reducing the development/modification time.

UI components are important constructs for encapsulating complexity and increasing productivity. A component is in itself a configurable, reusable minimal UI. It can be adapted by the developer according to the

context of use.

Usability might be addressed using components by encapsulating usability rules in these components. The developer can use the component out-of-the-box. An example of such components is: localizable calendar component (different calendars dates with different formats). However, components can only encapsulate usability rules at the concrete level of abstraction (the detailed design level). Abstract usability guidelines are impossible to support simply because they do not fit at the same level of abstraction covered by the approach. An example is: "cultural background affects the way people sort things". Such a guideline impacts the navigation design and cannot be embedded in the UI component.

2.6.2 Limitations

As we have seen in the visit to the history in section 2.2.1, the human factors crisis was reported in this approach. This is because it ignores the "what" and consequently cannot address the human factors.

Based on the adoption by the industry, we presume this approach is the most successful one at the time being.

However, even if we accept the promise of this approach, it suffers from different problems, that we enlist below.

1- Twisting UI code with other modules

A characteristic of this approach is that the UI is part of the development code. The code for the UI and the code for other UI modules are intermingled. No wonder to see some code that executes a specific functionality (or several) when the button is pressed. The same can be noticed in the other direction. It is common to see snippets of code that modify the UI while executing a function (create/remove/modify a widget).

In Java Swing, one can add an action listener to a button. When the button is pressed, the code in the action listener is executed. The developer can do whatever s/he needs there, including implementing a complete functionality in the system. This leads to complications when the interaction design is modified. Removing this button would remove the functionality behind. The same also goes for java methods that implement a specific functionality. They can interact with the UI by modifying or

even creating UI elements, like to display an error message.

This issue is usually addressed by coding conventions or architecture (like MVC: Model-View-Controller), where the developer aims at separating the UI from the functionality, and consequently have more freedom to manipulate the UI independently of the functionality.

2- No separation of concern

This approach defines everything at the same level. A button (a simple widget) is at the same level of abstraction as a drop-down box or even a grid/table. If we look closer, a button is not the same as a table. A grid can contain a button inside, but the opposite is not possible. Besides, a grid defines a tabular placement of its children elements, while a button does not have such a property. A grid may define navigation (pagination) when it has huge data to display. In java swing, the program will not complain if we substitute a button with a grid, as both are components in the program. Only the common sense of the developer prevents such a substitution.

The grid is not only at a different level of abstraction than the button, but it encompasses different concerns: navigation, placement in addition to stylistics.

3- Modularity may lead to repetition

Sometimes, the separation of concerns is addressed by modularity. A UI is separated into several modules, each with a specific purpose. An example is the web, where three modules are essentially used:

1) HTML: with the purpose of structuring the UI.
2) CSS: with the purpose of supporting stylistics in the UI.
3) JavaScript: with the purpose of adding behaviour to the UI.

The problem in these modules is repetition. The same activity can be carried out using different modules. For instance, JavaScript can modify the structure of the HTML page, thus conflicting with the purpose of the HTML. It can modify the styles as well and conflicts with CSS.

CSS can model part of the behaviour. For example, using CSS, we can hide parts of the page. Hiding parts of the page is related to the behaviour, and should be performed using JavaScript.

Repetition in modules degrades intellectual control of the application if

changes are made anywhere [Taylor2010, pp.40]. In the example on web technologies, modularity helped in separation of concerns, but degraded intellectual control by repetition.

4- Low traceability with requirements

We do not have any clue why a UI component is added on the UI. A button on the screen is there for a purpose. The approach does not provide any means to trace this element back to its roots, not at the design (if based on a design decision) nor at the analysis (if based on fulfilling a requirement).

Low traceability leads to difficulties when modifying the UI, a routine activity for developers after contacting the end-user. The developer needs to ensure that modifications of the UI do not have a side effect on existing requirements and design decisions already implemented.

2.6.3 Root cause of limitations

As we said earlier, this approach is the cause of the crisis. Let's discuss why it caused the crisis at the first place.

We think the root of the problem is related to the history of the evolution of UIs. At the beginning, the UI was concerned with technical aspects of the graphical card in the computer. It evolved later in the light of the WIMP (Window, Icon, Mouse and Pointer), which was also an evolution from a technical point of view. This evolution was happening in the light of: *how to develop a UI on the machine?* This question depicts a pure technical point of view to the UI domain.

The shift in the point of view happened with the human factors crisis. The point of view changed to: *how to design a UI for a user?* The shift happened in the minds of the HCI community and UI developers, but not in the UI itself. The technical point of view to the UI is still there, and is reflected in tools.

To make a change in the domain, we need to change the way we look at the UI to become aligned with the mentality we developed in HCI. We need a perspective to the UI to move it out from the graphical card (the machine) to the mind of the user, and then put it back on the machine. Maybe an approach is to re-analyze the interaction between the human and the machine, and then re-establish the technical UI based on this

new perspective.

Limitations mentioned in this approach are persisting since ever. If we could establish such a perspective, we may address all the concerns of this approach differently. A statement based on belief in the power of change.

2.7 The software development life cycle approach

The software development life cycle (SDLC) is concerned with developing a product following well-defined processes. As the UI development is part of the software development, SDLCs are ultimately concerned with developing a usable UI, and consequently, incorporating usability knowledge (guidelines) in their process.

In this section, we focus on how usability guidelines are integrated in different SDLC. We identify 3 types:

The Classical SDLC: these are the classical engineering approaches that follow a sequential performance of development activities through phases. The waterfall is the representative example of this type.

HCI-inspired SDLCs: After the emergence of the HCI, researchers proposed adapted SDLCs that explicitly address human factors in their processes. Examples are Curtis & Hefley Layered model [Curtis1994], Collin's Circle [Collins1995] and Nabla [Kolski1998].

Iterative and Agile methods: Agile methods are a common practice in software engineering. It is of much interest to see how they integrate usability knowledge and the UI development in their processes. We analyse them as part of iterative SDLCs.

2.7.1 Classical SDLC

Both the software engineering crisis and the human factors crisis were related to the classical waterfall SDLC. The cause of the human factors crisis is the non-consideration of the human factors since the early beginning of the project.

Figure 2.18 illustrates the problem in the waterfall SDLC in details. Firstly, it shows how usability spreads on all phases and is not addressed **explicitly** in any of them. Besides, it shows how the UI design is treated as a step of the detailed design phase.

The relations between the UI design and previous phases are bi-directional. The UI impacts the analysis phase and the analysis phase impacts the UI. The same goes for the design phase. Let's elaborate more on these bidirectional relations.

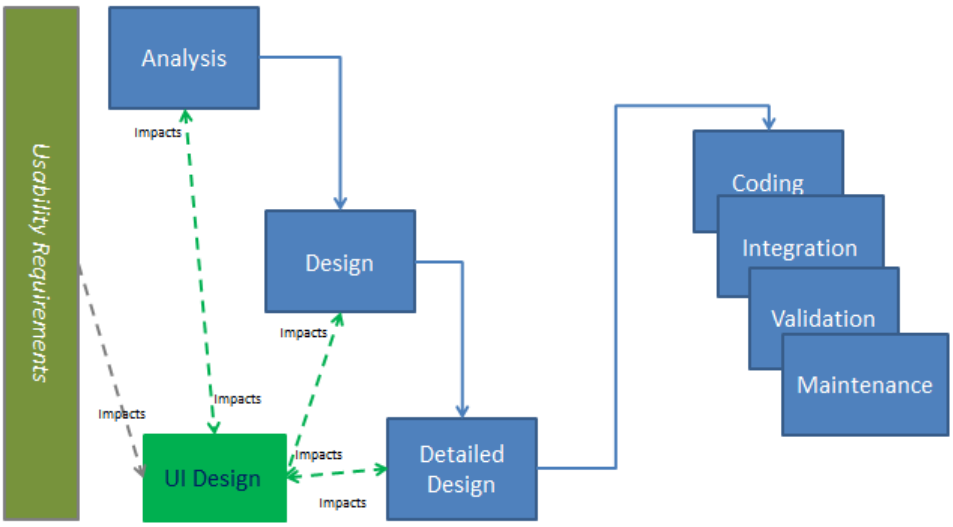

Figure 2.18 The bi-directional impact of the UI design and other development phases in the waterfall SDLC.

1- Analysis and design phases impact the UI

These are trivial relations, as it is impossible to develop a UI independently of a product. The UI design is part of the detailed design phase. It is based on design decisions and specifications coming from the design phase.

2- The UI impacts the analysis phase:

The UI impacts the analysis phase. Users and stakeholders are often, during the early stages of the project, unable to express and describe what their requirements really are. This is known as the IKIWISI syndrome [Boehm2000], short for: I cannot tell you, but I'll know when I see it. Therefore, the UI may help the user to describe requirements after seeing and/or using the UI. Therefore, the UI impacts the analysis phase.

3- The UI impacts the design phase

When designing a system, designers focus on key qualities (usability, security, performance, etc.). Software qualities are conflicting; we cannot design a system with all qualities supported. A tradeoff between these qualities is necessary. Henningsson's survey on the relation among software qualities as evaluated by the industry revealed that these qualities are dependent [Henningsson2002]. Enhancing one quality may affect others, either positively or negatively.

Note that to evaluate the usability of a UI, we firstly need to develop the UI. Consequently, modifications on the system to enhance usability might be carried out after the development of the UI. In turn, this might affect other depending qualities and result in modifications on the overall system design. Therefore, the UI impacts the design phase.

2.7.2 HCI-inspired SDLCs

1- Description

With the emergence of HCI and definition of HCI-related activities, the need to align activities of HCI and Software development led to a new type of SDLCs that are inspired by HCI.

Alignment is not merging. The UI has special characteristics that are related to human factors. Thus, the development of the UI should not be an isolated activity in the development process. It should start since the early phases of the software development life cycle and be aligned with the software development activities.

The promise of this approach is to re-define UI development activities in a process that is aligned with software development activities. This promise is depicted in the figure 2.19. The figure shows a generic set of activities. Each development process defines its own set of activities and mappings.

In the following, we present some development life cycles and show how they define their activities and mapping.

2- Comparing existing HCI-inspired SDLCs

For a SDLC to address human factors and usability, it should:

1) Address human factors since early development stages.
2) Integrate usability knowledge on different levels of details.

3) Address bi-directional relations with the UI described earlier.

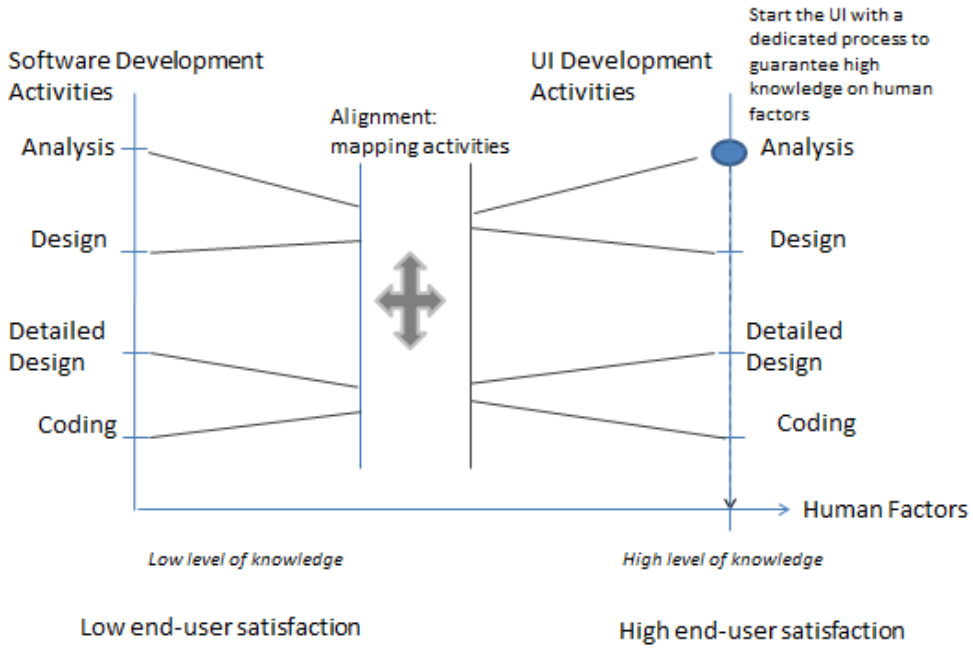

Figure 2.19 The promise of software development approach.

In the following section, we present and compare three SDLCs that are inspired by HCI.

The first SDLC is the Curtis & Hefley Layered model [Curtis1994] depicted in figure 1.20. This SDLC:

1) Defines explicit steps to address human factors in the development by defining explicit phases (on the left part in figure 2.20).
2) Usability guidelines can be integrated at different levels of details. For instance, guidelines related to the dialog design can be incorporated at the "dialog design" phase. If usability guidelines are classified after the phases definition, integration can be straightforward.
3) It does not address bi-directional relations with the UI. This is because we cannot have a UI until the coding phase. Usability evaluating can be started after we have a UI designed. If we have any modifications on the system design imposed by the usability evaluation, it cannot be addressed at a later phase (if any).

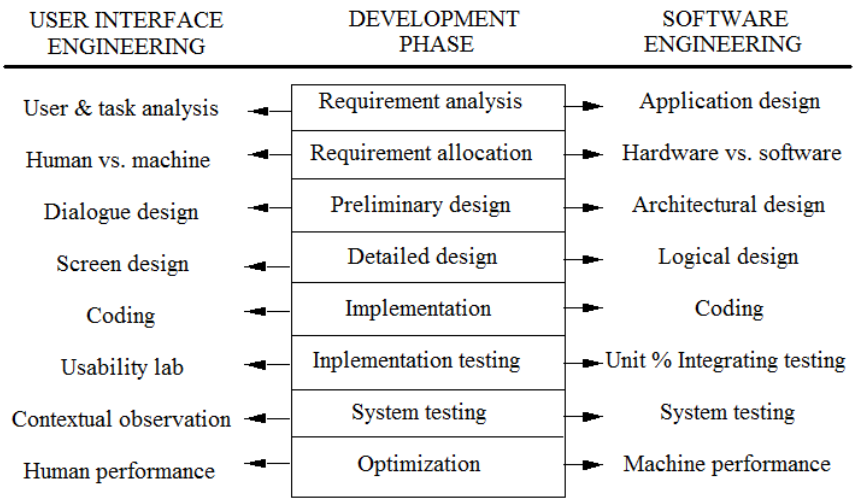

Figure 2.20 Curtis & Hefley's layered model [Curtis1994].

The second SDLC is the Collins Circle [Collins1995] depicted in figure 1.21. This model:

1) Defines explicit steps to address human factors by introducing phases: "User and task analysis", "Design user model" and "Design interaction and control mechanisms".
2) Usability guidelines can be integrated at the different levels. Probably in two phases (at least): the user and task, and at the design interaction and control mechanisms. We have fewer possibilities to refine guidelines than in the Curtis and Hefley's model.
3) It addresses the bi-directional relations with the UI at the circle level. We cannot have a UI until the "Prototype and evaluate" phase. To incorporate modification on the analysis and design phases, we have to wait until the next circle starts.

The third SDLC is the Nabla model [Kolski1998] depicted in figure 1.22. Simply explained, it consists of two Vs, one for software engineering (to the right) and another for HCI (to the left). These two Vs are aligned at the analysis and specification level, then parallel at lower levels. This model:

1) Defines explicit steps to address human factors introducing a set of phases to the left dedicated for HCI.

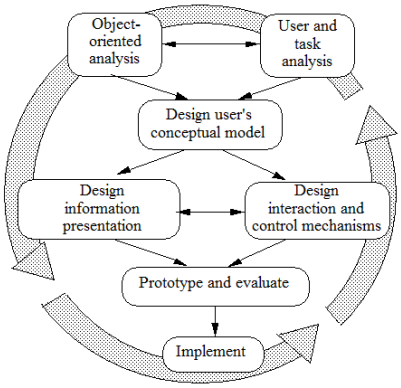

Figure 2.21 Collin's circle model [Collins1995].

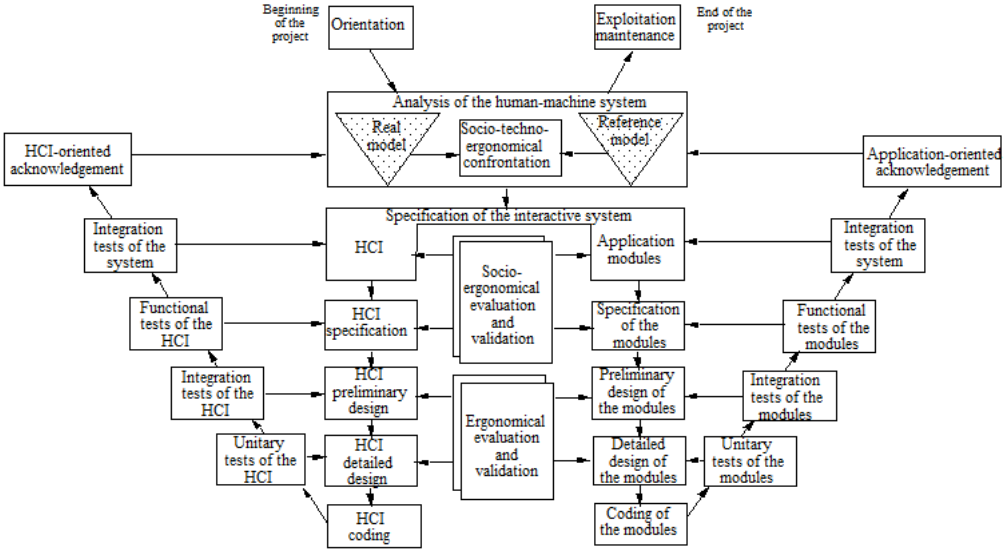

Figure 2.22 The Nabla model [Kolski1998].

2) It might be possible to integrate usability guidelines at different levels of details if we come up with a classification of these guidelines that can be mapped with defined HCI phases.
3) It addresses bi-directional relations with the UI at the V-level. We cannot have a UI ready until the "HCI coding" phase. To incorporate modification on the analysis and design phases, we have to wait until the next V is started.

3- Summary of the comparison

The table 2.3 summarizes the comparison between HCI-inspired SDLCs.

HCI-inspired SDLC	Address human factors	Integrate usability knowledge on different levels of details	Managing bi-directional relations with the UI
Curtis & Hefley Layered model	Explicit phases	High. By a classification of guidelines on defined phases	No
Collin's Circle	Explicit phases	Medium. By a classification of guidelines on two phases.	Yes. At the next cycle
Nabla	Explicit phases	Possible. A classification should be defined	Yes. At the next V

Table 2.3 Comparison between HCI-inspired SDLCs

2.7.3 Iterative and Agile Methods

Iterative methods address the software engineering crisis by developing the software through several iterations. Each iteration performs the complete stack of defined activities, including testing and getting the end-user's feedback. An iteration in the agile method explores requirements for the next one, towards a complete set of requirements in the final iteration. The last iteration denotes the completeness of the software product.

Iterative methods have a different promise to integrating HCI knowledge than other SDLCs. At the end of an iteration, a working UI is produced and the user can evaluate usability on that iteration. As iterations require short time to complete, the user can be contacted several times during the project and consequently, usability requirements can be gathered and integrated in the later iteration. Therefore, **the promise is "fast and repeated contact with the user":** develop a working (yet partial) UI fast and evaluate usability repeatedly since the early time of the project.

Figure 2.23 depicts this promise. It illustrates the UI resulting after each iteration, and how these UIs are converging towards the appropriate usable and final UI.

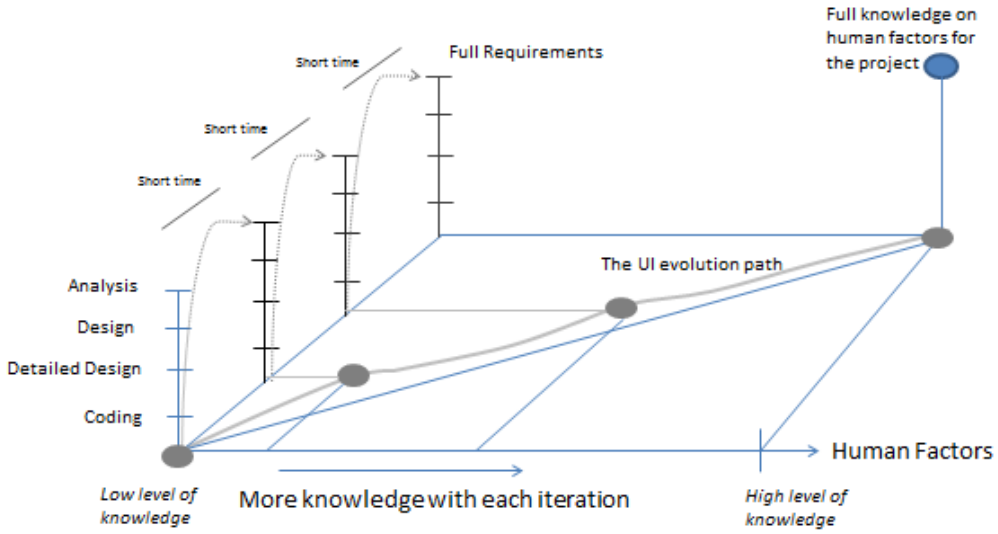

Figure 2.23 The promise of iterative methods to integrate UI development.

Agile methods are iterative and incremental methods. Examples of such methods are Scrum [Schwaber2001] and eXtreme Programming (XP) [Beck2004].

Notice that terms differ according to the method, even if they refer to the same concept. For instance, "iteration" in XP is called as "sprint" in Scrum. It is not in the scope of this thesis to discuss the details of these methods. We aim at understanding how the UI is developed in these methods and what the limitations are.

In what concerns the UI development, agile methods are interesting to integrate usability in the development. Their promise is realistic and proved practical with applied results. These methods focus on the concept of one team, where the user (or its representative) takes part. In Scrum, the role representing the user is called the "Product owner", while XP defines a "Customer" role to represent the user.

With the user part of the team, communication is enhanced, which ensures usability integration in the product. To make things better, methods that focus on usability, like User Centered Design, are integrated in Agile methods to ensure the focus on the user and usability. An Example of such integration can be found in [Deuff2012] and the new method is usually called a User Centered Agile Method. The interest of such meth-

ods can be depicted on the promise (in figure 2.23) as: grey circles (the UI after each iteration) get aligned on the shortest path towards the target.

1- Limitations

Although agile methods have been proved effective in the industry, there are still issues concerning integration of usability knowledge. Agile methods, whether employing a usability experts role in the team or not, design the UI one iteration ahead and validate its usability one iteration behind [Beyer2010]. More concretely, in iteration *i*, the role responsible for designing a usable UI start working on the UI of a component. S/he has all the time of the current iteration to do the work. In iteration *i+1*, developers start developing this component. Validation with the user can start after the end of this iteration, and preferably starts in the iteration *i+2* in order to respect the short duration constraint on iterations. Feedback and problems in the design are fixed in the iteration *i+3*. We call this problem as the **phasing problem in usability testing**.

The limitation that arises from the phasing problem is that usability testing might produce obsolete results. Some modifications on the product might have occurred during iterations between the UI design and the validation. These modifications may have impacted the UI design, and therefore, usability testing becomes non-significant.

The cause behind this limitation is related to the UI perspective adopted in these methods. The UI is a design aspect. Testing usability requires a working UI and should wait until the end of the UI development[26]. It is impossible to test usability in between these phases. Note that prototypes are design tools and cannot help in testing usability since the analysis phase.

[26] Prototypes are used in software engineering approaches to test usability. Prototypes are design tools, not analysis ones. To develop a prototype, we go through the same phases required to develop the real software product: analysis, design, coding and testing. Prototypes based on models follow the development phases of model-based approaches. Both approaches are discussed in this thesis and in both approaches the UI is a design aspect. Testing usability should wait until the end of the development. In this case, the end of the prototype development.

Again, this is the same cause identified in all previously discussed approaches. Assume a working UI (that can be executed) can be produced at each phase, so we can have: a UI at the analysis phase, a UI at design phase and so on until the final UI. In this case, the user can evaluate each UI focusing on different aspects, possibly during the same iteration. In this case, the UI evolves with the system development and does not lag behind due to the nature of usability testing.

2.7.4 The root cause of limitations of the software development approach

All the analysed methods in this section share the same traditional perspective to the UI development: a weak technical domain. The analysis gap is filed with cognitive models like the task and the user models. We have already explained the root cause of failure of integrating cognitive models in the UI development, and the same applies here.

However, if we could come up with a perspective that allows perceiving the UI at the analysis phase, the UI development would give a lot of benefits to the software development instead of the current situation. Assume one could build the UI at the analysis phase. The analyst can present the result of her analysis (or part of it) to the user and get his/her feedback on the real UI, not on a prototype. This mean, when the end-user agrees on the analysis phase, then it is sealed, applying the principle: what you see and sign on is what you get.

2.8 Summary

In this chapter we have analysed 5 approaches to integrate the UI and the software development. Our analysis revealed different shortcomings in each approach. It also revealed that these shortcomings are related to the same cause: the current perspective to the UI development considers the UI at the design level.

To address all stated shortcomings, and consequently enhance the ability to integrate the development of usable UI in the software product development, we need a new perspective that is more effective than the current one. Besides, the new perspective should allow establishing a life cycle that aligns the UI and software developments in a more effective way. Anyway, we discussed requirements for such a perspective in the

analysis.

In the coming section, we enlist shortcomings in different approaches and then set requirements to address/solve them. The mapping between requirements and shortcomings is stated later and depicted in figure 2.24.

1- Shortcomings

Table 2.4 lists shortcomings in the analysed approaches. It presents the hierarchical relationship between shortcomings to the root cause. These shortcomings are also depicted in figure 2.24 to give the reader a better understanding of them.

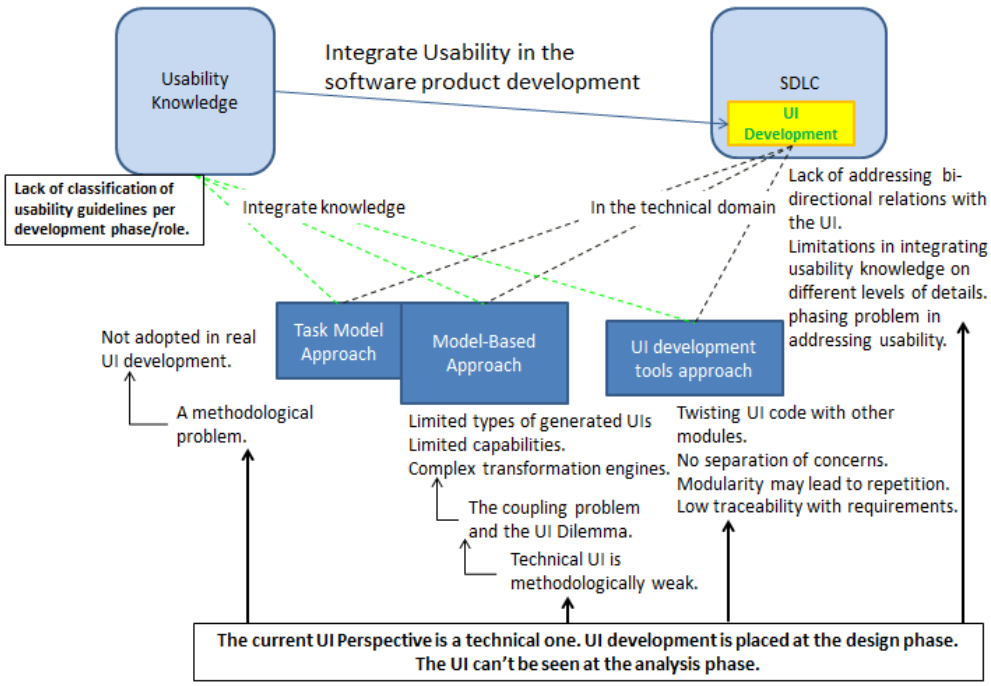

Figure 2.24 Shortcomings in the analysed approaches.

Approach	Short-coming ID	Shortcomings			Root Cause
		Level 1	Level 2	Level 3	
Usability Guidelines	UG_1	Lack of classification of usability guidelines per development phase/role			The current UI Perspective is a technical one. UI development is placed at the design phase. The UI cannot be seen at the analysis phase.
Task Models	TM_1	Not adopted in real UI development			
	TM_2	A methodological problem			
Modelling Approaches	MA_1	Limited types of generated UIs			
	MA_2	Limited capabilities			
	MA_3	Complex transformation engines			
	MA_4	The coupling problem and the UI Dilemma.			
	MA_5	Technical UI models are methodologically weak			
UI development tools	DT_1	Twisting UI code with other modules			
	DT_2	No separation of concerns			
	DT_3	Modularity may lead to repetition			
	DT_4	Low traceability with requirements			
SDLCs	SD_1	Classical and HCI-inspired SDLCs do not address bi-directional relations with the UI.			
	SD_2	Limitations on integrating usability knowledge on different levels of details			
	SD_3	Iterative and agile methods have a phasing problem in addressing usability.			

Table 2.4 Shortcomings in the analysed approaches

2- Requirements

To enhance the integration of usability in the UI development, we need to address shortcomings in table 2.4. Therefore, the first requirement is:

Req_1: Establish a new perspective to the UI development that:

Req_1.1: Enable the UI development since the analysis phase.

Req_1.2: Use concepts that belong to the technical UI domain only.

Req_1.3: Define different levels of abstraction/details, where

Req_1.3.1: A concept belongs to one and only one level.

Based on this new perspective, we need to establish a UI development life cycle that:

Req_2: Enhance the integration of usability knowledge on different levels of details.

Req_3: Address the bi-directional relations with the UI.

Req_4: Address the phasing problem when addressing usability.

To save thousand words, the mapping between requirements and shortcomings is depicted in figure 2.25. This figure depicts how the new UI perspective can address shortcomings in different approaches including the SDLCs approach. It establishes the logical mapping between requirements Req_1 (and sub-requirements) and other requirements.

Notice also how the hierarchical relation between shortcomings becomes a positive relation to address the shortcoming. Once the parent shortcoming is addressed/solved, children shortcomings are addressed equally.

In the coming sections, we establish a UI perspective that is based on a linguistic model. This linguistic perspective is the targeted one that we will use to establish the UI development life cycle.

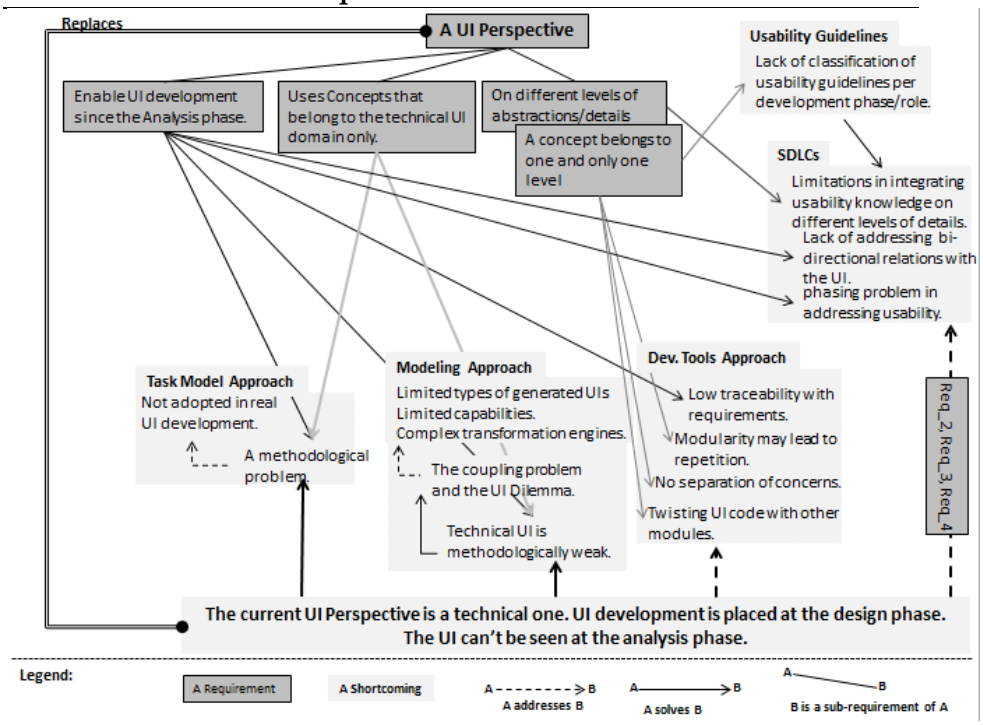

Figure 2.25 The mapping between requirements and shortcomings.

Chapter 3 A Linguistic Perspective to the GUI Development

3.1 Context

In 1986, Jacob Nielsen proposed a virtual protocol for computer-human interaction [Nielsen1986]. His protocol aims at analyzing and understanding the interaction between the human and the computer based on linguistic criteria. This chapter elaborates on his protocol to establish a linguistic perspective to develop GUIs. We introduce an adapted linguistic protocol for GUIs that is refined from the original work of Nielsen. The adapted protocol serves as a linguistic taxonomy of GUIs concepts, elements and artifacts: the underlying theory and methods to classify things.

The adaptation we need is to define a way to classify the GUI development activities and related concepts. We do so by analysing the GUI development activities thoroughly and placing each activity on the right level. This adaptation requires some modifications of the original protocol of Nielsen, which we present in coming sections. All along the adaptation, we present examples to explain how activities and GUI concepts are distributed on each level.

The linguistic perspective we are proposing is based on the evolution of the GUI. Assume we have a GUI that needs to be modified. If we can classify required changes on different levels, we can evolve the GUI per-level. This is explained further in the coming section.

Before moving forward, we may position our approach to a similar work that employs the linguistic model in the software development life cycle. Grislin *et al* employ a linguistic model in the V-Cycle development method in order to allow *a priori* evaluation of the software [Grislin1997]. They define a mapping between software development phases in the C-Cycle and activities and tools to use from HCI. The *a priori* validation at

the analysis phase is based on task models. Our approach moves a step further than theirs by aiming not only to evaluate the GUI a priori, but to start the concrete development of the GUI since the analysis phase. The interest of our approach is to ensure that what is developed and tested on the analysis phase is what the user will get in the final GUI.

3.2 Nielsen's Virtual Protocol for Interaction

The protocol consists of seven levels of interaction that are decomposed after linguistic criteria: goal, pragmatic, semantic, syntactical, lexical, alphabetical and physical. The HCI community considered this idea a powerful one at the time Nielsen introduced it. Nielsen aimed at analysing interaction between the human and the machine from a linguistic point of view to provide a better understanding of the domain.

The original idea of Nielsen lies in expressing interaction according to a linguistic classification benefiting from the following features:

1) Decomposition into linguistic layers: the interaction is decomposed into layers.
2) Communication between layers: each layer has two communication interfaces with other layers: an analyser interface with the upper layer, and a realizer interface with the lower one.
3) Concept univocal separation: each concept is univocally located in one and only one layer. This is the base for the principle of separation of concerns [Dijkstra1976], [Parnas1972].
4) Layer coverage: any interaction activity could be expressed with the full stack of layers

3.2.1 The virtual communication and the user interface

Communication between the human and the machine virtually happens at any level. In reality, it is happening on the physical level, as depicted in figure 3.1.

One characteristic of the interaction between the machine and the user is that it is controlled by the machine and guided by the user. For instance, the set of possible tasks to be carried out is determined by the machine. The user can guide the flow of the execution of these tasks by selecting one over another.

For the user to communicate with the machine, a user interface is re-

quired. This user interface must have input elements to allow capturing the user's input that guides the communication.

Assume a UI is developed following Nielsen's virtual protocol. The development of this UI would start from the top-level, refined on each level until we reach the lowest level. On each level, we would have part of the UI constructed[27] and refined. The UI on each level can be used in the virtual communication between the user and the machine on that level.

Communication at the physical level can be seen easily: the user uses the final UI, which is constructed iteratively on all levels. The virtual communication at the alphabetical level can be established by ignoring all physical-related characteristics from the constructed UI: light ignition of the monitor, physical movement of the mouse, electronic details to transfer information, etc.

How can we imagine the virtual communication at the task level? A simple answer is that the user should use the same UI at the alphabetical level after ignoring all details introduced at semantic, syntactical, lexical and alphabetical levels. If we remove all these details, what will remain of the UI?

An interactive UI needs input elements to acquire the user's part of the communication. Therefore, the minimal UI at the task level should have at least one input element. Besides, on the task level, we only have tasks (no semantic, syntactical, lexical, or alphabetical concepts). Therefore, the input element on the task level is limited to allow the user to interact with tasks defined on the system: tell the system to create, execute, repeat or rollback a task. In other words: the input element on the task-UI can only enable manipulating task, because there are only tasks at this level.

The same can be applied on virtual communication on lower levels. The UI at a level can only manipulate concepts that belong to that level. Besides, the UI at a level strips off everything related to lower levels, but it keeps what is related to upper ones.

In order to enable a UI development according to Nielsen's virtual pro-

[27] Constructed here is in the sense of: developed, produced, a working UI, a concrete UI.

tocol, we need first to identify what concepts belong to level, and what activities can be carried out on that level. This requires adapting the original protocol of Nielsen to enable such classification of concepts and activities.

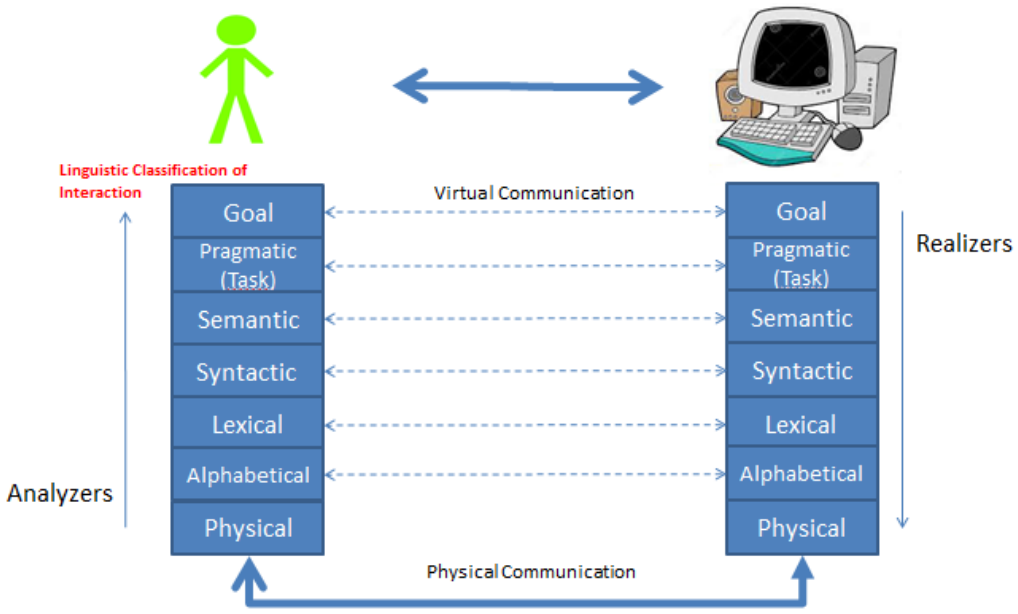

Figure 3.1 Overview of Nielsen's Virtual Protocol.

Before adapting Nielsen's protocol, let's first explain the linguistic perspective to the UI development in the coming section.

3.3 The linguistic Perspective to develop GUIs:

A linguistic perspective considers the interaction between the human and the computer as a discourse. It analyses this discourse as if it is a written text (the written text is the GUI) on seven linguistic layers, as defined in Nielsen' model: goal, task, semantic, syntactical, lexical, alphabetical and physical.

Let's look at the way an author writes an article in a journal. Each article should have a goal that the author satisfies through her written text. Each section in the article has a sub-goal. A section is divided into paragraphs (tasks), each serves a specific purpose (sub-goal). A paragraph consists of several phrases (semantic), each with a specific idea. Phrases

must follow linguistic rules to be syntactically correct in the written language (syntax). These rules apply on words constituting a phrase (lexemes). Words are formed of letters (alphabets).

How can we see a GUI (the graphical communication text) from the same point of view applied on an article? UI widgets (like a button, a text box, a label, etc.) are the smallest pieces we use to design a GUI. These **widgets** are **lexemes** (words) in the communication text representing the GUI. A widget consists of a set of properties that control how it appears on the screen. These **properties** are the **alphabet** used to write widgets. For instance, a simplified representation of a red label in the communication text might look like: {Type=label, background-color=red, etc.}.

Syntactical rules in human languages define the right sequence of lexemes for the phrase to be considered as correct in the writing language of the article. Syntax rules for a graphical communication text should define the correct sequencing of widgets on the graphical screen. These rules should define the **sequencing in space** (placement on the screen) and **sequencing in time** (the right moment to be presented on the screen). Human languages have pre-defined syntactical rules. Graphical communication texts do not have an agreed set of rules to define correct placements on the screen or on time. This is tackled in real life by defining best practices on the GUI design. With the lack of an agreed set of syntactical rules for the UI design, each GUI defines a unique and dedicated communication language.

Semantics in a graphical communication text can be seen in the grouping of required widgets to perform a **specific function**. This is to compare with the role of phrases in an article. A phrase is a set of tokens that adhere to syntactical rules and convey a specific idea to the reader.

A **set of functions** can help the GUI user to perform a **task**, which is compared to the grouping of phrases in an article to achieve a purpose. The **set of tasks** should allow the GUI user to satisfy his/her **goal**.

The above summarizes the way we look at a GUI from a linguistic perspective. Now let's adapt Nielsen's virtual protocol to use it as a tool to classify GUI concepts and UI development activities. In the coming section, we present examples and the next chapter demonstrates this perspective.

3.4 The Adapted Linguistic Protocol to GUIs Development

The adaptation we need is to enable the protocol to classify GUI development activities and related concepts. We will do so by analyzing GUI development activities thoroughly and placing each activity on the right level. This adaptation requires some modifications of the original protocol of Nielsen that we explain and justify inline.

The adaptation method works as follows: We firstly present Nielsen's definition of the level. Then we map the terms in the definition to GUI. We determine what should be identified at the level and what should be realized from the above ones. We provide examples when needed.

We exclude the physical level from the adapted protocol as it concerns the physical world, which is not part of our scope. The resulting adapted protocol is demonstrated in table 3.1. It consists of 7 levels: goal, task, semantic, syntax-time, syntax-space, widgets and widgets-properties. The "Adapted Levels" column denotes the seven adapted levels, while the right-most column denotes a broad distribution of key UI concepts on levels (the classification[28]).

Levels are adapted after an analysis to **linguistically classify GUI development activities**. Concepts are then identified at each level by analyzing the set of activities at that level, thus answering the question: **what concepts do we need to have on a level in order to carry out activities at the level in isolation from others?**

We explain levels and key concepts below (the linguistic classification). Key concepts (the right-most column) are defined after the explanation of the level depending levels. For instance, we explain the concept "UI Elements" after we explain the levels "Task" and "Semantic".

[28] We present a more detailed classification at the end of this chapter. The purpose of presenting the classification here is to help the reader to envision what kind of classification we will establish.

3.4.1 Goal

Nielsen's description: *"deals with the real world concepts that the computer system is all about."*

Goals at this level will be refined at lower levels to build the appropriate GUI. An example is when the user wants to reserve a flight on a web site. Real objects in this example include: the user, the flight and the ticket.

World[29]	Nielsen's levels	Adapted levels	Key GUI Concept[30] at the level
Conceptual	Goal	Goal	
	Task	Task	UI Elements
	Semantic	Semantic	
Perceptual	Syntax	Syntax-time	Containers and navigation elements
		Syntax-Space	
	Lexical	Widgets	UI Widgets
Physical	Alphabetic	Widgets Properties	
	Physical	*Out of scope.*	

Table 3.1 An overview of adapted levels and classification of key UI concepts

3.4.2 Task

Nielsen's description: *"This level deals with general computer-related concepts that are representations of the real world concepts from the goal level."*

This level realizes concepts at the above level by answering the question: what tasks are needed to achieve the goal on the above level? Note that several ways may exist to achieve the same goal. **This level is con-**

[29] Worlds are defined in Nielsen's Virtual Protocol.

[30] The way researchers abstract UI concepts depends on their perspective. The concepts presented here are abstracted based on the linguistic perspective.

cerned with tasks and their states and relations. Activities[31] that manipulate[32] the task model at this level should belong exclusively to this level. In other words: **lower levels cannot change the task model, relations or states, they can refine it only**.

3.4.3 Semantic

Nielsen's description: *"The detailed functionality of the system: exactly what each operation does to each object. There are a finite number of concepts in the system and they each have an exact definition."*

This level realizes concepts at the upper level (tasks) by defining detailed functions to carry out a task. Detailed functions can be of three types: (1) System Functions: carried out by the system. (2) Input Functions: require input from the end user. (3) Output Functions: display information on the screen.

UI elements: a UI element[33] is of type either input or output. UI elements are **concretized** on the screen as widgets. They can be visible (like a label, a text box or a buttons, etc.) or non-visible (like a finger gesture, a mouse click, a key-press). **Concretizing** input elements is an activity performed on lower levels.

UI Elements are identified at the task and semantic levels. The task level defines the tasks, transitions among tasks and states. Therefore, required UI elements at the task level are input elements to allow the user to perform such transitions and changes to the task state. For instance, a sub-

[31] Keep in mind the idea of virtual communication. Activities here depict the activities to carry out by the role involved in the virtual communication at this level (the user role or the system role).

[32] Manipulate a task by a role involved in the communication (the user or the system) is to carry out an activity on the task like to start, complete, reject, pause or resume a task.

[33] Well-informed readers on different types of abstractions of UI concepts may get confused with the way we abstract UI concepts from the linguistic perspective. We recommend them to compare our findings with what they know at the end of this chapter because the change in the perspective impacts the way to abstract concepts.

mit button in a login screen is the input element needed to change the login task state into a completed state. These input elements are identified at the task level as they manipulate the task model.

UI elements identified at the semantic level are related to the detailed function. An input function requires an input element, while an output function requires an output element. **UI elements at the semantic level** are different from UI elements at the task level in their **purpose and capabilities**. Semantic UI elements change the state of the related detailed function, but cannot change the state of a task[34]. This separation of UI elements on the task level from those on the semantic level is due to separation of activities imposed by the classification. Activities related to tasks should be carried out on the task level and not on any other level.

Assume we have a grid that displays a list of train trips from one city to another. The user needs to select the desired trip. **The input element to select the train trip** (could be a click on the desired row or a button/link on that row) is **a task input element** because it changes the current task's state (the "select a train trip" task). Assume that the user can click on a column's header (or a sorting icon in the header) to sort trips according to that column. The click on the column's header in this case represents an input element that is related to a detailed function (sorting). It shouldn't change the task state in anyway. The detailed function is added because the designer is willing to help the user to achieve the current task ("select a train trip") by providing the sorting functionality. The sorting input element (the header or the sorting icon) represents an input element at the semantic level.

3.4.4 Syntax-Time

Syntax-Time and Syntax-Space levels are adapted from the Syntax level of Nielsen's. This is because the description of Nielsen for this level is: *"Sequence of the input and output tokens exchanged on the underlying lexical level.*

[34] This is important to separate concerns between levels. UI elements identified at the task level can only manipulate concepts defined at the task level (tasks) Semantic UI elements can only manipulate semantic concepts.

The sequencing can be both in time and in space, including two-dimensional placement of tokens". The description denotes two types of sequencing: in **time** and in **space**.

Departure	Delay	Arrival	Delay	Duration	Connections	Type	Details
20:01		20:52		00:51	1	L,IC	
20:31		21:22		00:51	1	L,IC	
20:40		21:34		00:54	0	L	
21:01		21:52		00:51	1	L,IC	
21:40		22:22	+ 00:16	00:58	1	L,IC	

Figure 3.2 A grid showing semantic input elements in yellow. The task input element is accessible through a click on the row.

In a GUI, sequencing in time is (1) the grouping of UI elements inside containers, and then (2) the definition of the navigation from one container to another. These containers might be concretized later as windows or panels in a window, while navigation can be concretized as links or buttons (next and/or previous buttons) from one to the other. At a certain moment, several time-containers can be active, while others are not.

In a GUI, sequencing in space is (1) the placement of available UI elements inside containers, and then (2) the definition of the placement of these UI elements on the screen. A UI element is available if it is contained in an active time container.

Definitions of our levels are:

- **Syntax-Time level**: Distribution of UI elements in time. Represents distribution of UI elements on containers and defining the navigation among them. Examples of navigation actions are: press a link to open a popup, press a button to open a new widget. An example of a container is a table that contains output elements (the cells) that should be displayed together at the same time.
- **Syntax-Space level**: Distribution of UI elements in space. Represents the placement of syntax-time containers' UI elements on the graphical screen.

The syntax-time level realizes concepts identified at the upper level. That is, realize the grouping of these elements on time: what elements should

appear together and what elements should appear in sequence. Besides, it is responsible for defining and enabling navigation among these containers: none, one direction (navigate from one container to another) or bi-directional (navigate from one container to another and back forth).

Let's get back to the example on displaying the list of train trips. Upper levels defined all input and output elements needed. This level is responsible for grouping these elements together. Several possibilities exist: (1) show all elements together, (2) display most important information, and allow the user to see more on his/her demand. The latter solution requires changing sequence in time between less important information group and more important information group. UI elements in each group should be placed in a separate container, and a bi-directional navigation should be defined between these two containers. An example can be seen in figure 3.2 in the "Details" column. More information can be displayed but at a different time by clicking on the blue icon.

Enabling the navigation between time containers requires navigation elements. **Navigation elements** are input elements with a pre-defined behavior: navigate from a container to another. They are at a different level of abstraction from the input elements defined on the upper levels. To differentiate these elements from the UI elements defined at upper levels, we distinguish them as: navigation elements. However, navigation elements are treated as UI elements on lower levels.

To explain further the difference between input elements at levels: task, semantic and syntax-time, we refer to the linguistic perspective on an article. Phrases can be divided (using coma ',') to ensure the right sequencing of ideas. A period '.' denotes the end of a paragraph but might also refer to the end of a phrase. Input elements in a GUI can be used to complete a task or complete a detailed function. A navigation input element is like a coma ',' (that separates sequences in a phrase), it separates time containers and ensures the right sequencing on time.

Syntax-time containers are logical groups of UI elements that should be available together at the same time. Availability of a UI element on the screen here is not related to visibility of the concrete widget. A UI

element might be concretized as a non-visible input (keyboard shortcut or gesture).

3.4.5 Syntax-Space

The syntax-space level realizes time containers identified at the upper level by answering two main questions. The first question is: how to position concurrent active containers on the screen? While the second question is: how to position UI elements of an active time container on the screen?

This level realizes syntax-time containers by defining placement of concurrent syntax-time containers (syntax-time containers that appear at the same time) on the screen.

A syntax-space container is a group of UI elements that belong to concurrent syntax-time containers. A syntax-space container defines placement rules that control its UI elements placement on the screen. These rules define acceptable placement of UI elements on the screen. On the other side, these rules cannot determine the exact placement because we do not know at this level how UI elements will be concretized. For instance, assume a "search for information" task. This task requires an input element "A" to launch the task, and at least one input element "B" to fill in search keywords (user's strokes are inputs). A placement rule at the syntax-space level can look like: *place B after A*. If "A" is concretized later as an invisible event, the rule is considered to be satisfied. If A is concretized as a button, the button is ensured to be placed after the UI element B. Figure 3.3 shows an example from a university website where the search button is concretized as an event (the user must press the "Enter" key to launch the search) not as a concrete visible widget (a search button).

Displacement elements are input elements with a pre-defined behavior: allow navigation on the visible screen. They are on a different level of abstraction from input elements defined above. They allow navigation inside a syntax-space container to make some parts of the GUI visible. An example of these input elements is a scroll-bar. However, these input

elements could be concretized differently on lower levels: like a scroll-bar, a hand gesture on a touch-screen, a button/link in a web page to move to the top or the bottom of the page.

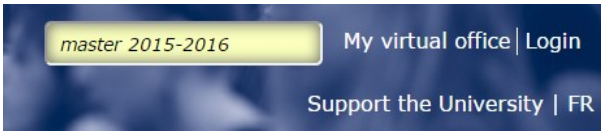

Figure 3.3 A search task implemented in a university web site with the search input element concretized as an event "press Enter"

3.4.6 Widgets

This level is called the lexical level in Nielsen's protocol. Nielsen's description for the lexical level is: *"using the information-carrying symbols of the interaction, called tokens. A token may be a word or a keyword, a special symbol, an icon, a number, a pair of screen coordinates, etc. These symbols are the smallest units that have their own system-related meaning."*

This level is adapted according to the linguistic perspective to the GUI development. Concrete widgets are the lexemes. They are smallest units carrying information in the interaction, and are used to write graphical phrases. Therefore, we disagree with Nielsen considering screen coordinates as lexemes. Screen coordinates have a meaning to the active window. A window has a meaning in the graphical system. A point *(x,y)* does not look to us meaningful without a graphical component.

This level realizes UI elements (including syntax-space containers) that are defined in upper levels by mapping them to GUI widgets. A simple example on this level can be seen in a login screen. A login is a task that defines an input element, which can be concretized at this level as a button (usually the case). If a user name and a password are only needed for login, we need two input elements (related to input functions), which might be concretized as a text box widget for each.

The mapping between UI elements and widgets is not always a one-to-one. A widget might be mapped to several UI elements. This is the case in the example we presented at the semantic level on a grid with sorting functionality. In that example, clicking the header of a column

sorts the grid. In this example, the header of the column is a label widget that is mapped to two UI elements: an output element (the column title), and an input element to execute the sorting detailed function.

3.4.7 Widgets Properties

This is the alphabetical level in Nielsen's protocol. Nielsen's description of this level is*: "Any system contains an alphabet of primitive information-carrying units, called lexemes. They do carry information, but they do not have any meaning by themselves. Only when they are put together to realize tokens can they be interpreted … in system terms.*"

Widgets are objects. Properties are the alphabets of an object. They represent primitive information-carrying units (the state of the object).

Setting the color and the font of a UI widget belong to this level. Translation of text from a language to another fits on this level too. Translation here maintains the same meaning of information, but changes the representation. Conversion of units and colors representation (RGB to Hexadecimal) all fit on this level. Coordinates of a widget fit also at this level.

Changing properties of UI elements can have consequences on upper levels. For instance, a large enough value for the width attribute of the username widget can prevent placing the password widget next to it and might break a syntax-space rule. Although possible to happen, it is detectable by the analyzer from lexical to syntax-space.

3.5 Summary

Table 3.2 summarizes our linguistic classification of GUI concepts and activities.

Table 3.2 depicts the linguistic categories in the second column. We call them levels. On each level, the linguistic classification defines a set of concepts that belong to the level. The communication interface among levels is well defined: what shall be defined and what shall be realized on each level.

World	Level	Artifacts	Concepts	Communication interface	
				Realize from upper level	**Define for lower level**
Conceptual	Goal	Goal		-	-Define goals and real objects
	Task	Tasks	UI Elements	Goals	-Define tasks and relations amongst -Define task input elements.
	Semantic	Detailed functions: System, Input, Output		-Realize tasks by defining needed detailed functions.	-Define input and output elements
Perceptual	Syntax-time	Time containers. Navigation elements	Navigation	-Realize distribution of UI elements on time by defining time containers.	-Define time containers -Define navigation elements.
	Syntax-space	Space containers. Placement rules. Displacement elements	Placement	-Realize placement of UI elements in time containers on the screen.	-Define space containers -Place UI elements on space containers - Define displacement elements
	Widgets	GUI widgets	GUI Widgets	-Concretize UI elements by mapping with appropriate widgets.	-Map to appropriate concrete widgets for UI elements
Physical	Widgets Properties	Properties of GUI widgets		-Realize widgets attributes.	-Set attributes of widgets -Defines nothing for lower level

Table 3.2 The Linguistic classification of GUI concepts and activities.

The linguistic taxonomy (the underlying theory of classification) is defined after the linguistic perspective and applied in section 3.4 to produce two classifications:

1) The linguistic classification of GUI concepts: illustrated in columns 2 and 3 in table 3.2.
2) The linguistic classification of GUI activities: illustrated in the last column in table 3.2.

Based on this linguistic perspective, we can move further to employ modeling techniques on each level and define a linguistic model for the GUI. From these models, we establish a linguistic development life cycle for the GUI. But before that, and in order to move on a solid ground, we need first to validate that the linguistic perspective is the one we are seeking in our requirements.

In the next chapter, we validate the linguistic perspective ability to develop a GUI, and to fulfil requirements on the perspective we are looking for. The next chapter also elicits linguistic modeling requirements, and therefore, establishes the bases for the chapter after on linguistic modeling.

Chapter 4 Demonstrating and Exploring the Linguistic Perspective

4.1 Context

In this chapter, we will:

1) Demonstrate the linguistic perspective through a case study.
2) Explore the linguistic perspective further by analysing more examples.
3) Elicit linguistic modelling requirements in preparation for the next chapter.

The case study is a **proof of concept** that demonstrates the linguistic perspective. It also serves as an internal validation of the requirement Req_1 "*Establish a new perspective to the UI development"* and its sub-requirements.

4.2 A Case study on developing a GUI linguistically

In this case study we illustrate the UI on each level and how activities are divided among levels. The case study is developing a GUI for registration to a conference.

The description of the case study is: the end-user fills the registration information and then pays the fees for the conference. Registration information includes the user's personal information, registration type (regular, student or discounted fees), additional information (if exists) and billing information.

This case study is well known in conference sites. A final GUI may look like Figure 4.1. We ignored the payment part because the same that applies on the registration can be carried out on the payment part.

The case study is a reverse engineering of the GUI in figure 4.1 to proof that the same GUI can be developed from a linguistic perspective.

4.2.1 The goal level

The goal of the GUI is: Register for a conference

4.2.2 The task level

On the task level, we realize the goal by performing two tasks on the task level: *Fill registration information* and *Pay conference fees*.

The minimal GUI at the task level will contain an input element to carry out each task. The two tasks identify two task input elements: "Finalize order" to complete the first task, and "Pay" to complete the second task. In reality, only one of these input elements is enabled at a time. However, enabling/disabling/hiding task input elements is the responsibility of the task model at this level. For simplicity, we show them both in figure 4.2.

4.2.3 The Semantic level

On the semantic level, for conciseness, we only refine the task "*Fill registration information"*. At this level, we need to define all the UI elements needed to carry out the task. Detailed functions required by the task are a mixture of input and output. They include all personal information, conference fees among others. In figure 4.3, we show to the left the final UI for registration to a conference. The figure depicts on the right how this UI would look at the semantic level. Notice the green box on the right, which depicts the task input element that is defined at the task level. The "Pay" task input element is omitted from this figure to show it is not active yet. Grey boxes in figure 4.3 represent output elements, while white boxes represent input elements. Notice that the decision to concretize them as labels and text boxes is not taken at this level.

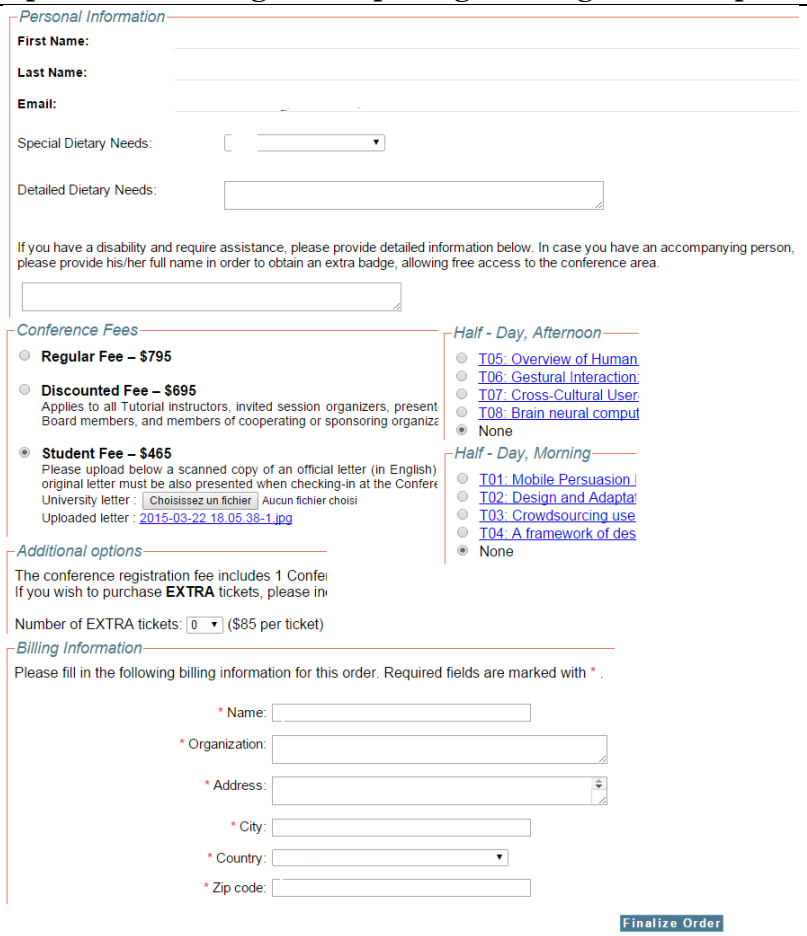

Figure 4.1 The final GUI of the case study.

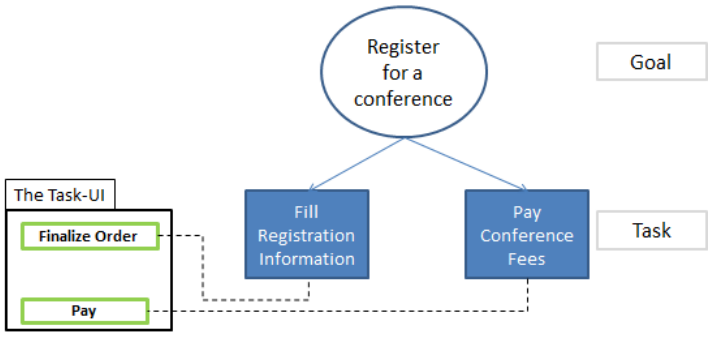

Figure 4.2 The UI on the task level

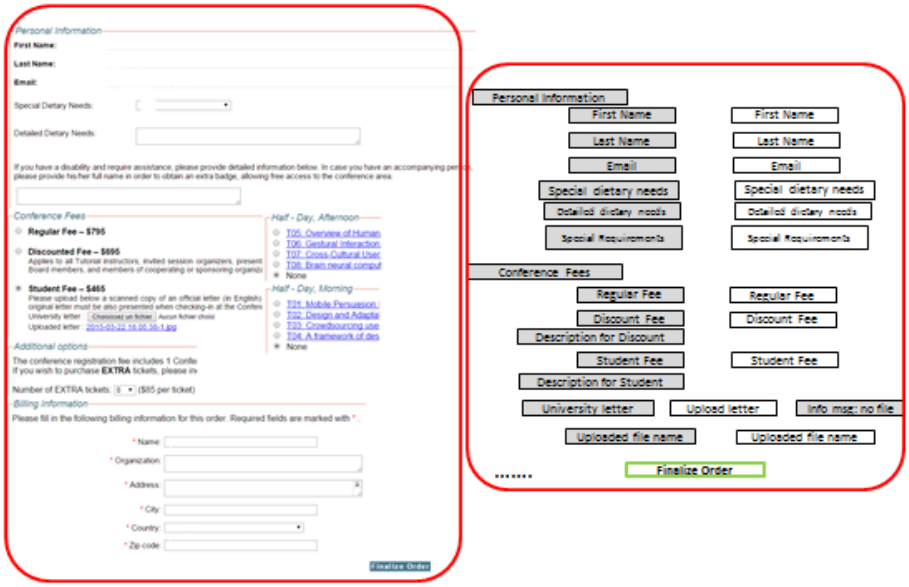

Figure 4.3 The UI on semantic level.

4.2.4 The Syntax-time level

On the syntax-time, we define distribution on time. We may have two styles:

1- Style 1: Display everything together at the same time. Therefore, we make no changes on figure 4.3.
2- Style 2: Define navigation as depicted in figure 4.4.

Figure 3.6 depicts the UI at the syntax-time level, following style 2. It illustrates how to distribute UI elements on time-containers and define navigation among them. We show portions of the final UI in some containers to allow the reader imagine how the final UI will be constructed later. Anyway, the first two time containers depict how the UI is displayed at this level. The reader should ignore the placement or elements inside these containers, as this will be defined on the lower level. Note the definition of navigation elements: "Next step" and "back". These navigation elements are defined based on the selected style of navigation. Note also how UI elements form upper levels are distributed between time containers.

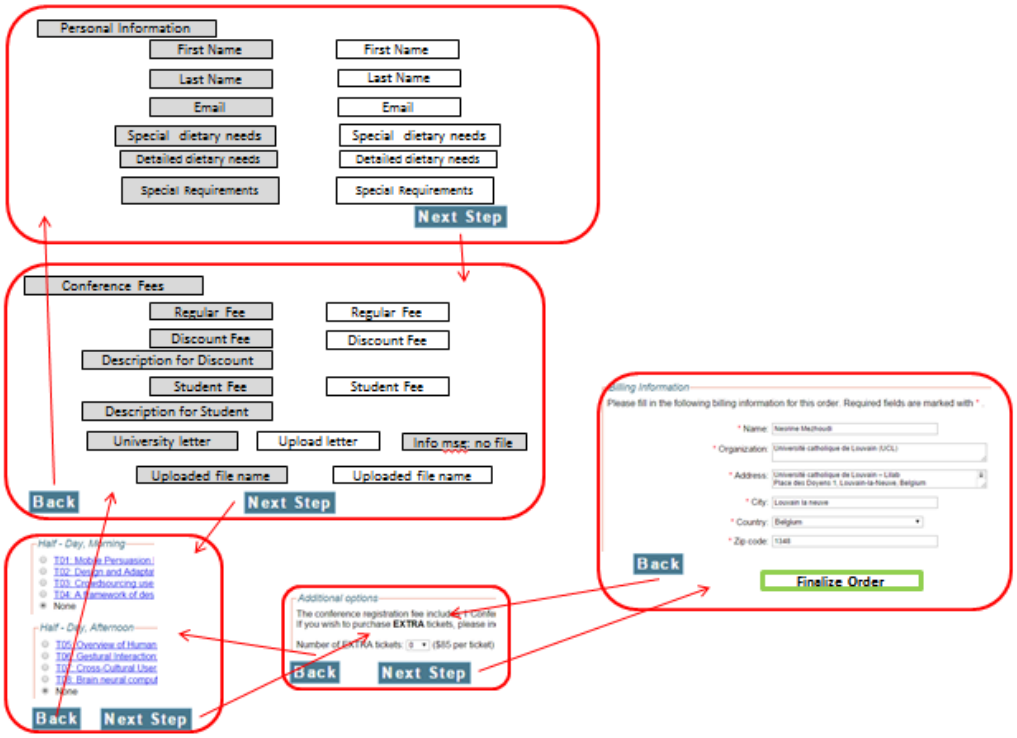

Figure 4.4: The UI on the syntax time: time containers and navigation elements are defined.

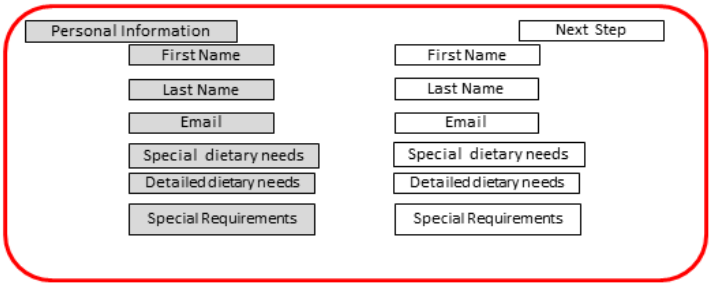

Figure 4.5: The UI on the syntax-space level.

4.2.5 The Syntax-space level

On the syntax-space, we refine only the first time container in the syntax-time style 2. The refinement defines the placement of active time containers on the screen. Figure 4.5 depicts this refinement on this level

for the first time container: "Personal Information". Notice the vertical alignment of UI elements on two columns.

4.2.6 The Widgets level

On the widget level: map UI elements to concrete GUI widgets as in figure 4.6. The illustrated mapping is one-to-one. However, it is possible to have many-to-many type of mapping, if we have complex widgets. We will analyze complex widgets and how to map them in a later section.

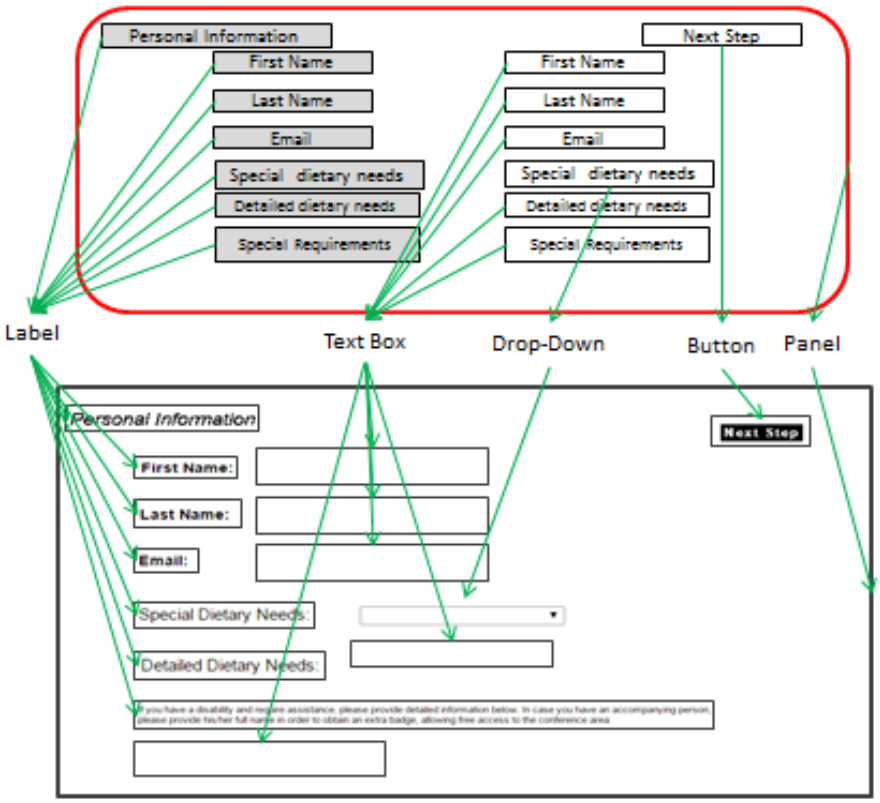

Figure 4.6 Mapping of UI elements with concrete GUI widgets on the widgets level.

4.2.7 The Widgets Properties level

Finally, on widget Properties level: We set properties of widgets to get the final GUI like in figure 4.7.

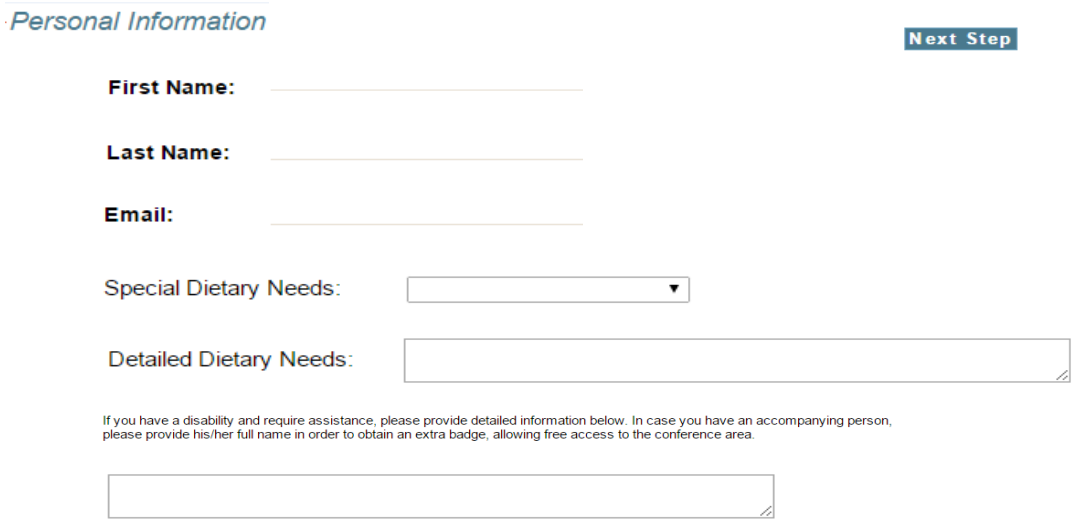

Figure 4.7 The UI on the widgets properties level.

4.3 Internal validation of requirements

This section validates that the linguistic perspective fulfils requirements we identified in chapter 2.

4.3.1 The linguistic perspective satisfies requirement Req_1

The first requirement we identified in chapter 2 is to find a linguistic perspective with specific characteristics stated as requirement Req_1 and sub requirements.

The linguistic perspective satisfies all the requirements we are seeking. We proof it in the following.

1- Req_1.1: Enable the UI development since the analysis phase

Since the early beginning of the project, the UI is built gradually. We start from the task-UI that controls the way to carry out the tasks. The task-UI is a working UI and can be presented to other members of the project team and to the end user to gather their feedbacks.

The task UI contains task input elements. It allows the user to test a pos-

sible order of performance of tasks.

2- Req_1.2: Uses concepts that belong to the technical UI domain only

All the concepts in the linguistic perspective belong to the technical domain. The main concepts are: UI elements (input and output), Containers (space and time), Widgets and their properties. The reader should not confuse the task level, the "task" concept and the "task input element". The task concept defines task input element at the same level of abstraction.

The linguistic perspective extends the technical domain to cover the analysis phase. It does so by employing a different abstraction than common. Abstraction on linguistic levels of technical UI concepts reveals that not all input elements are at the same level of abstraction. The linguistic perspective identifies: task-input elements, semantic-input elements, navigation elements and displacement elements.

3- Req_1.3: Defines different levels of abstractions

The UI is divided into several linguistic levels. Each level refines concepts from the upper level by adding more details.

UI development Activities and UI concepts are classified linguistically on these levels. A level represents also a category in the linguistic classification. The linguistic classification is used as the abstraction point of view.

The linguistic classification has the property of "concept univocal separation". The adapted linguistic classification inherits this property to univocally separate UI concepts on linguistic levels. This satisfies the requirement Req_1.3.1: "A concept belongs to one and only one level".

4.4 Exploring the linguistic perspective

We argued in chapter 2 that addressing a root cause would highly impact the problem and other superficial causes. In this section, we explore the linguistic perspective to assess the impact on the usability guideline approach.

4.4.1 A classification tool for UI modifications

The linguistic classification has the property of "layer coverage". The

adapted linguistic classification inherits this property and therefore, it allows classifying any aspect of interaction on one of its levels. This would create a tool to classify any kind of changes to the UI based on required activities and affected concepts. Let's present some examples.

1) *Translate text inside a label from English to any other language*: Alphabetical level.
2) *Change the color, font or background of a label*: achieved by setting an attribute on the UI element. Thus, it is at the alphabetical level.
3) *Replace a button with a hyperlink (in web platform)*: lexical level.
4) *Change placement of UI elements from vertical to horizontal, from left-to-right to right-to-left:* Syntax-space level.
5) *Place an element on the specific position (x,y) on the screen*: Alphabetical level.
6) *Splitting UI containers*: re-distribution of UI elements on UI containers with probably re-definition of navigation model. Syntax-time level.
7) *Replace a combo-box with a text box or vice-versa*: This is a change in the definition of the data type of a UI function. The combo-box accepts a value from a list of values displayed to the user. A text box accepts free text. Besides, a drop-down box defines a vertical placement of its values in addition to a predefined set of styles. It is at several levels starting from the semantic level to the alphabetical one.
8) *Add/modify/delete an attribute on the data model*: Semantic level.
9) *Add a new button*: this depends on the purpose of the button. We need to understand the change to determine the type of the input element behind.

Eventually, when a level is changed, lower levels may need to be changed as well.

The linguistic classification of changes is similar to the effect of a prism on the light. The prism disperses the visible light to the spectrum light. The linguistic classification disperses a change to activities on levels. Eventually, non-GUI changes are dispersed away, just like the prism does with invisible lights.

We borrow the prism metaphor from optics to represent the linguistic classification, which we call: a linguistic prism. This is depicted in figure 4.8.

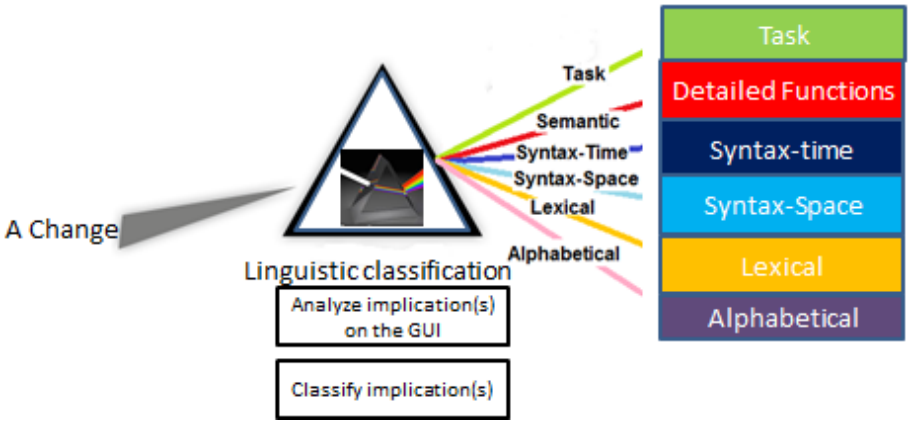

Figure 4.8 The Linguistic Prism

4.4.2 A potential classification tool for usability guidelines

In our discussion on classification of usability guidelines (2.3.6), we argued on a classification based on the nature of guidelines as an alternative to a role-based classification. The aim of such a classification is to integrate usability guidelines in the development. We stated some requirements on the classification, we discuss them here.

1) Have a countable and reasonable number of categories: The linguistic classification has 7 categories.
2) Reflect the level of abstraction of the UI concept: The linguistic classification abstracts UI concepts on 6 classification categories (levels).
3) Classify all UI concepts: the linguistic classification classifies all UI concepts.
4) Univocally classify UI concepts: This is satisfied in the linguistic classification.
5) UI activities defined should be conceptually independent from each other: This is also satisfied thanks to the linguistic classification of GUI activities.

Therefore, the linguistic perspective is a potential classification of usability guidelines to incorporate them in the development. It can classify the change embedded in the guideline, using the Linguistic Prism.

The interest of such a classification is to incorporate guidelines in the UI development. The linguistic perspective develops the UI through re-

finements on different levels. A linguistic classification of guidelines identifies the appropriate levels to incorporate them in the development.

Mapping linguistic levels to development phases and roles (thus establishing a UI-DLC) would turn the linguistic classification of guidelines fruitful, because it would **fulfil the shortcoming on classification of guidelines (UG_1)**.

Let's give three examples on classifying guidelines.

- *Cursor movement should be minimal*: This guideline impacts the placement of UI elements on the screen. Therefore, it can be employed on the syntax-space level only either manually or automatically.

- *Error messages should provide a specific feedback*: This guideline impacts the semantic level by either adding an output element to display the error message or impacting the content of the message itself.

- *Colors should be distinguishable*: This guideline impacts the color concept that belongs to alphabetical level (see section 4.4.1).

4.4.3 Support the evolution of the UI

The linguistic perspective enables the development of the GUI in a refined way: from high-level concepts to lower ones. At each level, a working UI can be developed.

The UI development can be seen as an evolution from one step (before a modification) to another (after the modification). The first working GUI evolves from the void step (no UI). Modifications evolve the UI to another step where these modifications are implemented.

The linguistic perspective enables the GUI to evolve per-level. If a change requires a modification to one level, upper levels are not affected. Therefore, the UI can evolve on lower levels only.

The evolution per-level impacts maintainability of the GUI. When modifying a level, we are sure that upper levels are not changed. Compare this to a GUI that is developed using a GUI framework. In the latter case, changing the color of a widget may impose the need for a full test to be sure nothing is changed unintentionally. This is because everything is put together. The linguistic perspective requires testing only the modified level and below, thanks to the separation of concerns. For instance, when we substitute a widget with another, we are sure the new UI fol-

lows the same tasks like the previous one, the same semantics, navigation and placement on the screen.

Figure 4.9 illustrates how the GUI can evolve per-level due to changes in the context of use. The evolution happens by extending the GUI using fragments of code (instructions) per-level. Note that such an evolution allows older versions to keep working (backwards compatibility).

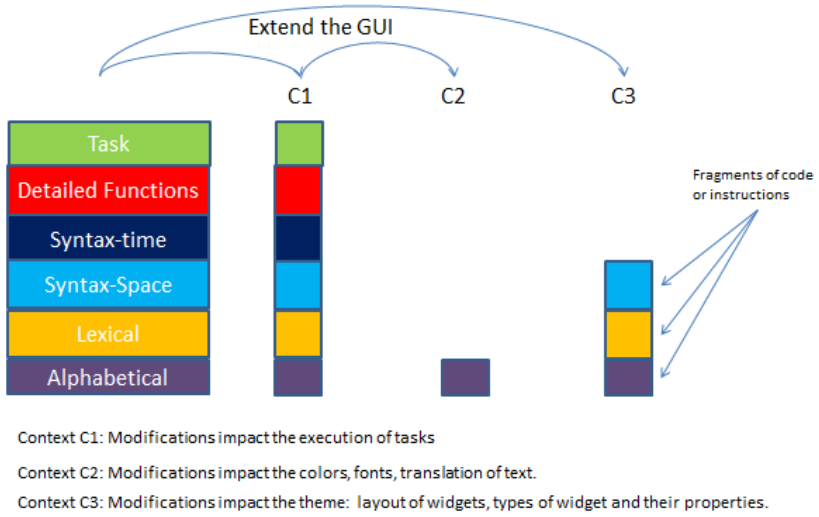

Figure 4.9 Evolution of the GUI by extensions

4.5 A Linguistic Modeling of GUIs: Eliciting requirements

Existing models for the UI address the reality where the computer, the designer, the context of use and the user are the principle objects and actors. It is expected to have concepts representing these objects and actors in models.

The linguistic perspective addresses the same reality, but primarily focusing on classifying GUI development activities. With regards to the constraint to isolate activities univocally on one and only one level, we may say that the linguistic perspective is **slicing GUI concepts and development activities into levels**.

The slicing of concepts has an impact on how to do modeling from a linguistic perspective. A concept is defined on a level, but refined later on lower ones. The behavior of objects (operations defined on these objects) belonging to this concept is also repartitioned on levels. Operations that can be executed on a level cannot be executed on another. For instance, defining a task input element can be executed solely on the task level. If another level allows this operation, the perspective is challenged. Therefore, **modeling has special requirements per level**.

The slicing of concepts and their behavior does not impact the overall abstraction of the concepts. If we merge all slices together, we would get the same concept as defined in common modeling approaches. Therefore, we do not have a GUI model per-level. In fact, we have a GUI model that is sliced onto different linguistic levels. Therefore, what we have on a level is a slice of the GUI model. The term: **modeling on a level** refers to apply an abstraction on the slice of the GUI model on that level.

In this section, we aim to elicit linguistic modeling requirements on levels, by analyzing different examples. Although the analysis presented here in non-exhaustive, it sets a base and a common understanding to how to analyze activities and classify GUI concepts and artifacts on further cases. Therefore, we analyze some key activities when developing GUIs. These activities are:

1) Adding/modifying/deleting an input element.
2) Complex widgets.
3) Hiding/enabling and disabling of UI elements.
4) Positioning of elements on the graphical screen.

Before starting the analysis, let's have a look at the linguistic modeling framework.

4.5.1 The linguistic modeling framework

The linguistic perspective defines the communication interface between levels as depicted in table 3.2. This table can give an overview of how the GUI concepts and artifacts are sliced. This in sequence can give an overview of how levels can be abstracted and interrelated. Figure 4.10 depicts

the interfacing between levels. The main benefit is the ability to abstract a level independently from others while respecting the interfacing protocol. The figure illustrates that we can have at the same time two working models at a level: one that realizes one part of the elements from the upper level and the other realizes the other part. This property enables the use of two different abstractions or notations on the same level, which may have an impact on the expressiveness and/or flexibility of the slice on that level.

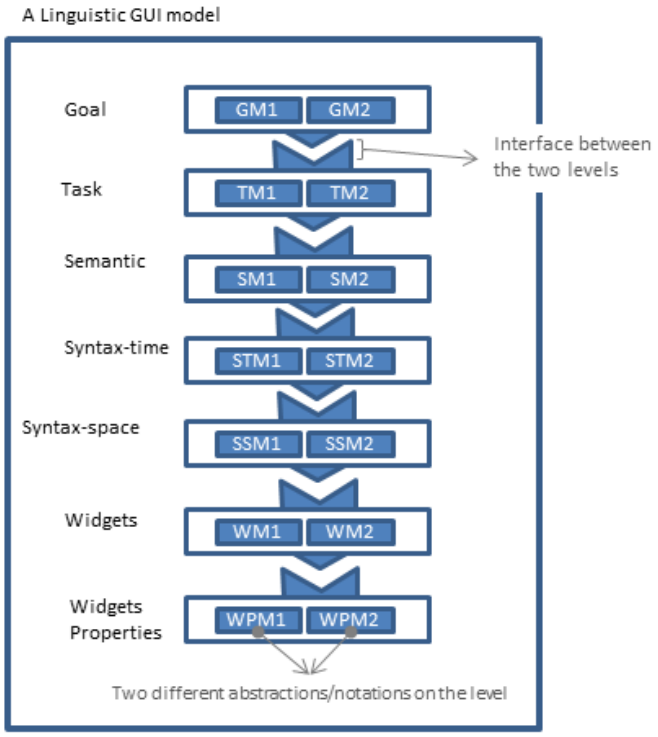

Figure 4.10 The Linguistic GUI modeling framework

With the modeling framework presented, we start analyzing examples to elicit modeling requirements per-level.

4.5.2 UI Input elements

We already mentioned that the linguistic perspective abstracts UI input

elements on different levels of abstractions with different purposes for each level. However, it might be strange that the only level that does not define any input elements is the widgets level.

We think the widget level should not allow adding or removing an input element. Although it is the level that handles concrete widgets, we should not allow adding widget freely at this level. Adding widgets breaks the traceability between the conceptual and the perceptual worlds. A widget has a purpose on the screen. This purpose should be traced back to the appropriate level that defines it.

The following requirements should be considered in any linguistic model:

1) The widget level shall not allow adding an input widget.
2) The widget level shall allow mapping UI elements from the upper level with widgets.
3) Input elements identified at the task level are exclusively eligible to change a task state.
4) Input elements identified at the semantic level are exclusively eligible to launch a detailed function.
5) Navigation elements (Syntax-time) are limited to enable navigation from a time container to another.
6) Displacement elements (Syntax-space) are limited to enable navigation on the screen, inside the space container.

4.5.3 Complex Widgets

Complex widgets spread over several linguistic levels. They do not fit on the widget level, as one would expect. Let's see this on two examples: a grid and a combo box.

1- The grid widget

Analyzing a grid that displays tabular data from a linguistic perspective leads to a classification on several levels. A grid on the semantic level looks as a large group of output elements: an output element for each cell. Every output element is related to detailed system functions that realize a task on the upper level (mostly a task that displays the list).

On the syntax-time level, if we ignore pagination, these output elements are grouped into one time container, as they will all be displayed at the

same time. On the syntax-space, we define rules to place these output elements in rows and columns. Every output element should be related the row and column the element belongs to. This requires a sort of logical grouping for output elements. The semantic level is the appropriate level to define such logical groups, because it knows the semantics behind these output elements. Thus, a requirement on the semantic level is:

7) The semantic level shall allow defining logical groups for UI elements

2- The pagination on a grid

Let's get back to the syntax time level to assume the grid allows pagination: it displays ten rows per page. If the number of rows is reasonable, enabling pagination on a grid can be performed on the syntax-time level: create a time container for each ten rows and define the desired navigation.

If the number of rows is large, it is preferred to implement the navigation of the data, not on the UI for performance issue (retrieving thousands of records from a database to display them by 10s on the UI can cause a performance bottle neck). When the user clicks on the next page, the system will fetch only the requested ten rows. In this case, navigation elements become detailed functions at the semantic level. One time container is enough on the syntax-time level.

The above example illustrates that visual appearance might be misguiding. In both types of pagination, the UI would look the same. The linguistic perspective illustrates the difference in the UI structure. We can talk about semantic pagination and lexical-time pagination, which conveys a precise meaning.

The example also illustrated that time containers and navigation elements are generated at run time, depending on the number of output elements defined (produced) on the semantic level.

For the above, we elicit the following requirements:

8) Levels shall be treated at run-time.
9) The syntax-time level must place UI elements from the semantic level, at run-time, on time-containers.

3- The Combo-Box widget:

Let's compare two complex GUI widgets that look similar: a combo-box and a list box (see figure 4.11).

The list box is populated with data. This data is displayed on the screen as a list of clickable elements that are contained in a container. Thus, each element in the list is an input element. The list box is a container of input elements that are generated on the semantic level for each piece of data. The same applies to the combo-box.

The difference between the combo-box and the list box relies in the need to click on the combo-box to open the list of choices. Input elements in the list box are displayed together at the same time in the containing window. Therefore, we have only one time container on the syntax-time level. The combo-box displays elements at a different time than the containing window. Therefore, we have two time containers: the first is for the containing window while the second is for the elements in the combo-box. The click to open the combo-box is a navigation element

from the first container to the second.

Figure 4.11 A list box and a combo-box widgets

Notice that on the syntax-space level, both widgets employ a vertical placement rule for their elements.

The examples on the grid, the list box and the combo-box demonstrate that complex widgets spread over several linguistic levels. If we intend to use them on the widget level, we need to map appropriate UI elements to them. For instance, the click on the combo-box should be mapped to a navigation element not to another type of input.

4.5.4 Hiding, disabling and Concrete UI elements

1- Hiding a UI element

The term "hide" looks ambiguous from the linguistic perspective. Different activities on different levels can lead to visually hiding an element:

- Moving a UI element from an active time container to an inactive one hides it from the screen.
- A space container can hide UI elements from a space container beneath.
- On the alphabetical level, we can hide a widget by giving it a color similar to the background.

All the above cases are different from removing a UI element, which we call: un-defining a UI element. Un-defining a UI element has the same visual effect as hiding. The main different between both is that the visually hidden element is still there, but the un-defined element is not.

Hiding the element at the semantic level does not un-define the element, therefore, it should stay accessible. Attention to this property when implementing a semantic model should be considered. If hiding is meant as removing, it should be implemented as un-defining, or else the element will stay accessible. In fact, this could be an interesting criterion to validate a navigation model: the navigation model should allow accessing all UI elements generated from top levels.

Hiding an element by placing it in the background space container might be considered as an error. This is critical if the hidden UI element is a task input element. But sometimes, UI elements are intentionally hidden, like when displaying an advertisement that fills the screen. However, it is certainly an error if the advertisement does not have a close button (or a means to remove it). Overall, it is possible to detect overlapping between UI elements and issue a warning.

From the above, we elicit the following requirements:

10) The level that defines a UI element is responsible of un-defining it: this is important to isolate concept-related activities on the level that defines the concept.
11) The widget properties level shall not have a property to hide a widget: this is to avoid the visual ambiguity on the widgets level.

2- Disabling UI elements

Disabled (paused) tasks are not necessarily realized as disabled UI input elements. A disabled task might be hidden on the GUI. The appropriate place for disabling input elements, just like the case of hiding, is where we define them. The same applies for other levels. Therefore, we elicit the requirements:

12) The level that defines a UI element is responsible of enabling/disabling that element.
13) The widget properties level shall not have a property to enable/disable a widget.

3- Widgets

The widget level realizes UI elements by mapping them to concrete GUI widgets. This realization should respect the type of the UI element. For example: an input element should not be realized as a displayed label. Output elements are elements that do not allow the user to perform any kind of input. They cannot be mapped to buttons.

A button in java allows several inputs: a click, mouse over, mouse-down and mouse up. It also accepts an output: the caption. Because a GUI widget may have multiple inputs and/or outputs, it can realize several UI elements.

For example, in a simple login screen, that contains a text box for the user name, another to the password and a button to submit. The UI designer may remove the login button and submit the form when the user hits the 'Enter' key inside the password text box. The later GUI is identical to the former on all levels from the task till the syntax-space. The difference relies in the mapping of two input elements (the login and the password) to one widget (the text box for the password).

On another side, replacing a UI element by another should guarantee to realize all input and output elements realized by the previous one. We can expect also that a GUI widget might be replaced by several widgets or the inverse (several widgets replaced by only one).

From above, we elicit the requirements:

14) Widgets shall have a meta-description of inputs and outputs it can map to.
15) UI elements relation with GUI widgets is of type many-to-one.

4.5.5 Placement rules versus explicit coordinates

Placement of UI elements is an activity on the syntax-space level. The coordinates of the widget are classified at the alphabetical level. This could lead to ambiguity in classifying the activity of placing elements on the screen.

To solve this ambiguity, we clearly define the role of each level. The syntax-space defines placement rules, not absolute positioning on the screen and these rules should be respected by lower levels. Therefore, the coordinates of a widget are relative to the containing widget and are validated at run-time to respect syntax rules.

From above, we elicit the requirements:

16) The coordinates of a widget are relative to the containing widget.
17) The coordinates of a widget are validated against placement rules at run-time.

4.5.6 Summary of modeling requirements

From the analysis above, we summarize the elicited requirements in Table 4.1. Requirements are categorized per-level. Note that these requirements are elicited based on the nature of the linguistic modeling. They are interpreted as: *for a model to be considered a linguistic model, the following requirements should be satisfied.*

4.6 Summary

In this chapter, we demonstrated the linguistic perspective on a case study that demonstrated a proof of concept of the ability to develop a GUI from a linguistic perspective. We also validated that the linguistic perspective is the perspective we are seeking, because it fulfils requirements on the perspective identified in chapter 2.

We explored further the linguistic perspective and its implications on the guidelines approach. The linguistic prism is a potential candidate to classify usability guidelines in a way that enables their integration in the UI development. We presented some examples on how to classify guidelines to support this argument. The linguistic prism classifies guidelines based on the type of change imposed. Figure 4.12 depicts our progress on the requirements chart graphically. The degree of the green colour depicts

the level addressing: the darker, the more profound.

Level	Requirement
All	-Levels shall be treated at run-time.
Task	-Task input elements are exclusively eligible to change a task state. -Exclusively allow un-defining/enabling/disabling task UI elements. -Task UI elements may not be refined to visual elements on UI. They might be concretized as events on some visual widget on the GUI.
Semantic	-Semantic input elements are exclusively eligible to launch a detailed function. -Allow defining logical groups on UI elements. -Exclusively allow un-defining/enabling/disabling Semantic UI elements. -Semantic UI elements may not be refined to visual elements on UI. They might be concretized as events on some visual widget on the GUI.
Syntax time	-Allow defining navigation elements. -Navigation elements are limited to enable navigation from a time container to another. -Must place all UI elements from the semantic level, at run-time, on time-containers. -Exclusively allow un-defining/enabling/disabling navigation elements. -Navigation elements may not be refined to visual elements on UI. They might be concretized as events on some visual widget on the GUI.
Syntax space	- Displacement elements (Syntax-space) are limited to enable navigation on the screen, inside the space container. -Exclusively allow un-defining/enabling/disabling displacement elements.
Widget	-Do not allow adding an input widget. -Allow mapping UI elements from the upper level with widgets. -Shall have a meta-description of possible inputs and outputs it can accept. -UI elements relation with widgets is of type many-to-one.
Widget Properties	-Shall not define a property to hide a widget. -Shall not define a property to enable/disable a widget. -The coordinates of a widget are relative to the containing container. -The coordinates of a widget are validated against placement rules at run-time.

Table 4.1 A summary of linguistic modeling requirements

In order to employ modelling from a linguistic perspective, a list of linguistic modelling requirements is elicited. In the coming chapters, we establish a linguistic model that fulfils these requirements.

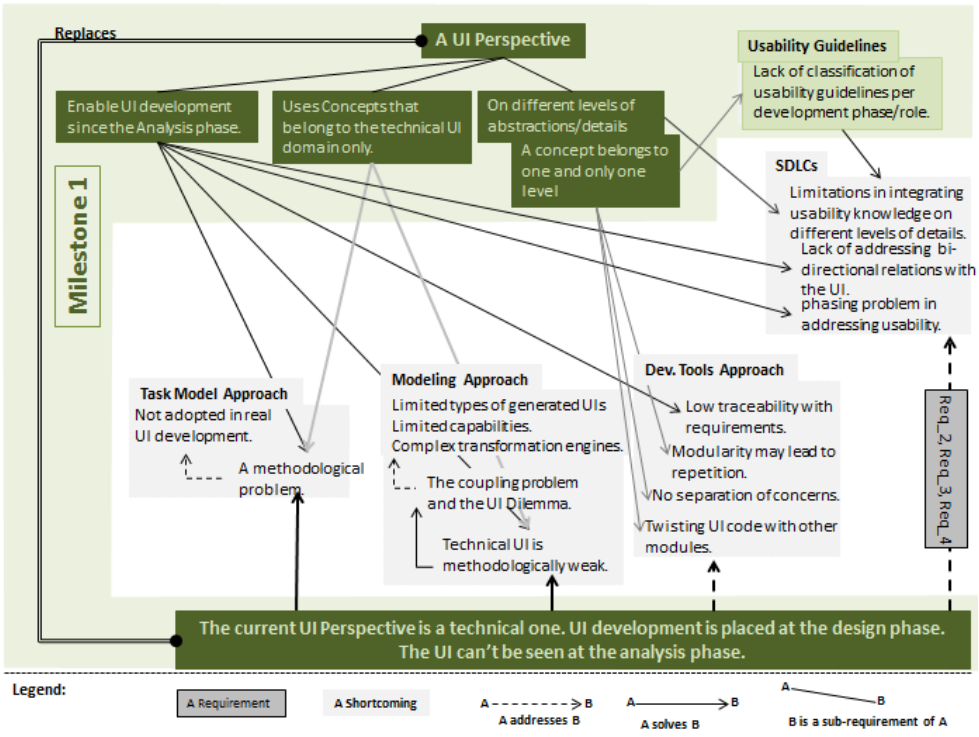

Figure 4.12 The progress map at the end of the first milestone.

Chapter 5 A GUI Linguistic Model: The Task Level

5.1 Context

This chapter is the first part of three chapters on the linguistic modelling. In these chapters, we model some levels and, then move forward to implementing and validating these models.

Validation is made on the same case study that we introduce when validating the task model. This allows the reader to have a better understanding on the working of these models and how the GUI is refined recursively.

The purpose of these chapters is a proof of concept that a linguistic model of the GUI is possible. We propose a model on each linguistic level, which represents a slice of the linguistic GUI model. A GUI linguistic model is the sum of different models on all linguistic levels with respect to the communication interface among them.

The set of proposed models here are not exclusive. Other models could be proposed on each level for different purposes. The important constraint on linguistic models is to respect the linguistic modelling framework presented in chapter 4.

5.2 The goal and the task model

The first step towards a linguistic model is to model the goal and the task levels.

For a task model to conform to the linguistic modelling framework, it needs to have:

1) Well-defined criteria to separate between the goal and the task levels/models.
2) Well-defined criteria to separate between the task and the semantic model/level.

3) A notation that supports identifying task input elements.

Our proposed task model follows the hierarchical task analysis approach. Unfortunately, hierarchical task analysis couples goals and tasks in an inseparable way. The hierarchy of tasks is a hierarchy of goals and sub-goals [Annett1971]. Anyway, we do not know of other approaches that separate goals and tasks clearly. Thus, the first requirement to separate between goals and tasks is impossible to satisfy. Our proposed task model merges both levels together.

The linguistic modeling framework imposes a task model on the analysis phase, and a strict separation from the lower phases (the semantic and lower levels). Therefore, it **solves the methodological problem** in UI task model (refer to chapter 2) **by limiting the role of the task model on the analysis phase**. As discussed in chapter 2, this solution requires searching for a Task Decomposition Stopping Criteria (TDSC) that is independent from the design concepts and the technical UI domain. The second requirement is concerned with finding a TDSC that is independent from the technical UI domain.

The third requirement is addressed by introducing a task-modelling notation that enables automatic identification of task input elements.

Let's first handle the TDSC and then move to the notation and the linguistic task model.

5.2.1 A Design-Independent Task Decomposition Stopping Criteria

As we have seen in chapter 2, existing UI task models employ a TDSC that depends on the design of the UI. Therefore, none of them can apply in our model. However, further research revealed fruitful results for our target, as we came across Guerrero' model integrating UI design in workflow systems [Guerrero2008].

Guerrero's model contains three main entities: the workflow, the process and the task. The workflow is related to a hierarchy of processes. Leaves in the process hierarchy represent tasks. The CTT model is employed to model these tasks and consequently the UI.

Guerrero's model solves the methodological problem in UI task models by limiting the role of the UI task model to the design space. The analy-

sis phase is covered by the workflow and process models, while the design space is covered by CTT. Although we doubt the expressiveness of CTT to cover the UI design space, the approach is interesting because it employs a methodological way to separate the analysis from the design, by identifying leaves (root tasks) in the process hierarchy based on well-defined criteria.

The root task identification criteria in Guerrero' model are based on changes on the work environment as described in the scenario that describes the work. A complete description of these criteria can be found in appendix A. Below is a short description:

- *Change of space*: when the location of operations changes.
- *Change of resource*: when the scenario suggests that new or different resources are exploited. Resources are of types: user, material and immaterial
- *Change of time*: when the scenario indicates a different time period in which the task is performed. Three types exist: interruption, waiting point (decision or accumulation) and permanence of execution unit (synchronization point).
- *Change of nature*: tasks can have the following natures: manual, automatic, interactive or mechanical.

Our linguistic task model is intended to cover the analysis phase. It ends where Guerrero's task model starts. **Our idea is to use the same criteria to identify leaf tasks in the task analysis** in contrast to *identify root tasks* in Guerrero's model. In this way, the task analysis can stop decomposition when no change is justified among decomposed or children tasks.

Guerrero's TDSC is independent from the design space and the technical UI domain. Therefore, it is appropriate to use in our task model.

5.2.2 Limitations in Task notations to identify task input elements

The widely used notation in tasks models is the CTT notations. Although several variations for this notation exist to enhance expressiveness power of the task model or mapping with system models, like HAMSTERS [Barboni2010], they all use temporal relations defined in CTT.

Although powerful, CTT temporal relations are not enough to identify

task input elements in the linguistic perspective. We show a scenario on this limitation.

In order for a user to search for a flight, s/he fills in search parameters. The system searches for relevant flights and displays them on the screen. The user selects the preferred flight. This scenario can be modelled using CTT notation as in figure 5.1.

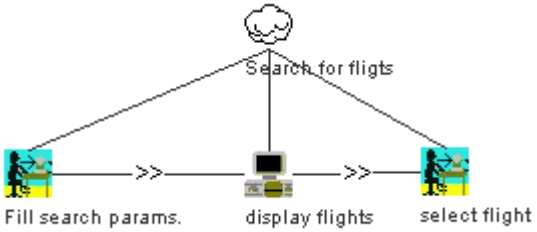

Figure 5.1 Search for flight using CTT notation

The tasks "display flights" and "select flight" in figure 5.1 might be implemented in the GUI as a table of flights with a "select" button on each row to select the flight. This **select button is a task input element** because it completes the task "select flight". The number of task input elements is equal to the number of flights in the search result. The linguistic perspective requires identifying all task input elements, provided that we know the number of flights to display. The notation does not reflect that the **"select flight" task will produce a number of task input elements related to the number of flights displayed**. This dynamic aspect in task execution is important in the linguistic perspective.

Note that another style is to display one flight and a select button at a time. This style differs from the above at the syntax-time level: defining time containers and navigation. Even in this style, we need to identify all produced task input elements.

K-MAD notation provides a solution to this dynamic aspect, but the notation introduced employs calling functions on the data model. From a linguistic perspective, these functions are classified at the semantic level. If such functions are to be used on the task level, they should take the form of a service to be implemented on the semantic level. Klug [Klug2005] introduced a task state and ports on tasks to exchange data to turn CTT model into executable task model. Data ports from the linguistic perspective are at the semantic level. Anyway, task state diagram

is interesting and at the linguistic task level.

In the next section, we introduce a linguistic task model notation that supports producing task input elements.

5.2.3 A Linguistic Task Model

The need for a new task model notation is justified by the linguistic requirement to identify task input elements. Task analysis is already explained in the literature on task models and we have already adopted TDSC from Guerrero model.

The task hierarchy is established by logically grouping identified tasks (like in CTT). A task can have one of the five categories: user, interactive, system, mechanical and abstract. These categories encompass categories in CTT and in Guerrero's model. The change we introduce in our task model is a notation that supports identifying task input elements.

Identification of user input elements is based on the task category. User tasks are not considered in the GUI, because they are manual tasks. Thus no input elements are needed for these tasks.

System tasks are performed by the system and require no interaction from the user to be started or completed. The same applies to mechanical tasks that are performed by machines (not computers), so they have no impact on the interaction.

Abstract tasks are logical tasks in the hierarchy and do not have effect on identifying task input elements (they group input elements from children tasks). The only interesting category that affects identifying task input elements is the interactive category as they are performed by the user using the GUI.

An interactive task needs at least an input element so the user can denote the task as **completed**. Other task input elements might be needed to give the user control over the performance of the task. For instance, can the user rollback a completed task? This is important if the user can change her mind after completing the task (an example is selecting a flight and then deciding to select another).

1- The task-state diagram

Determining needed task input elements for a task depends on the task state. The user needs a complete input element on the GUI to complete

the task in hand. If the task can be suspended, a suspend input is needed. Once it is suspended, we may need to resume it explicitly using a different task input element. Our task model defines a configurable task state diagram for tasks. Each task can configure its diagram. **Identification of task input elements for an interactive task depends on its configured state diagram**. The generic task state diagram is depicted in figure 5.2. The states are explained in table 5.1.

Transitions in a task's state diagram from state A to state B is defined as a condition. When the condition is satisfied, the transition occurs. There are three types of conditions:

1- Automatic: the condition is always true and the transition is automatic
2- User: the condition is true when the user provides a required explicit input.
3- Other Condition (or Condition for short): any other Boolean expression. When this condition is true, the transition is performed.

State	Type
Created	The task is created by the system. Task initialization can happen here.
Offered	The system offers the task to the user to start
Started	The task is in the course of running.
Suspended	The task is in the course of running. For interactive tasks, an explicit input from the user is needed.
Completed	The task has completed execution and the subgoal is satisfied.
Destroyed	Resources reserved by the task can be released
Errored	Something prevents running the task.

Table 5.1 Definition of task states

A transition type is the type of its condition. Transitions for system tasks can be of type "Automatic" or "Condition". User and mechanical tasks have no impact on the GUI. However, for consistency, a state diagram with "Automatic" transitions is attached to user and mechanical tasks. Transitions for interactive tasks are more sophisticated.

Employing task states can be noted in other researches, like in

[Gharsellaoui2014], where researchers enrich CTT by adding states to tasks to support run-time execution of the task mode. Task states in our model enable identifying task input elements. Besides, researchers in [Gharsellaoui2014] use CTT temporal relations to define transitions among tasks, while states in our model play a major role in defining these transitions (explained in the coming section).

2- Properties to configure an interactive task state diagram

Every interactive task must define the condition for the complete transition state. This condition must be of type "User" or "Condition", with "User" as a default. Setting the condition type to "User" means the task requires an input element on the GUI to allow the user to complete it. Setting the condition type to "Condition" means the task depends on the state of another task(s) to complete. We elaborate more on this case after introducing relations between tasks. An interactive task sets, by default, all transitions to "Automatic", with the exception for the complete transition. Transition types can be configured by setting the task properties. These properties and their impact on the state diagram are discussed in the following section.

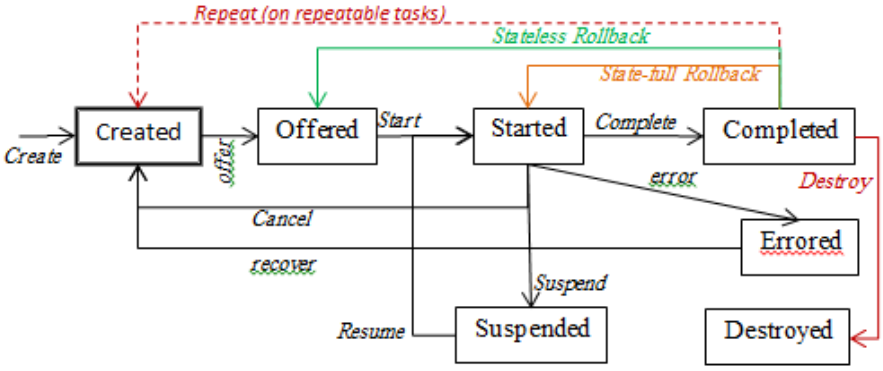

Figure 5.2 The Task State Diagram and Transitions.

Assume a user performing a task would like to withdraw already filled in information and restart from the beginning. Such tasks are cancellable tasks. Setting the canCancel task's property to true sets the cancel transition type to "User". This impacts the GUI by adding a task input element to cancel the task. Setting the canCancel property to *"false"* removes the cancel transition from the task's state diagram.

Assume the user can rollback a task. If the task can be rolled back, a task input element to rollback is needed on the GUI. An example is when the user selects a flight then changes her/his mind. Not every task can be rolled back. A payment for a flight task cannot be rolled back by simply changing the task state. Rolling back a payment usually requires a different process. Rolling back a task can be of two types: (1) Stateless: re-create the task and discard all previous task information (2) State-full: restore the task information previously entered before completion. The task property canRollBack can have one of three values: false, stateless and stateful. The value of this property controls the rollback transition. The false value removes the rollback transition. The stateless and stateful values set the rollback translation type to user, although it can be modified to a condition (a relation to another task).

The property "mayError" denotes a task might be prevented from execution due to various circumstances. If these circumstances are detectable by the system, we can define a condition on the error transition. If these circumstances are not detectable by the system, the user should be given this means to move the task to the error state by a task input element on the GUI. The "mayError" property can have one of the values: false (removes the error transition), condition (define a condition transition) and User (define a "User" error transition). Another related property is the "canRecover" which denotes how to recover from the error if possible. It accepts the same values of the property "mayError".

The property "canSuspend" on a task controls the suspend transition in a similar way to canCancel property. The resume transition is influenced by this property. A task might be suspended by the user action, but resumed based on the state of another task. In this case, the suspend transition is set to user but the resume transition is a condition. The resume transition initial type is set to user when the suspend transition type is set to user.

The destroy transition is a special case and is defined when a task is dynamically created by another. Dynamic tasks are discussed later in this section. For dynamic tasks, destroy transition can have one of two types: user or condition. Offer and start are condition transitions. The user cannot control to start a task by an input. The system offers and starts the task once conditions are satisfied. The create transition is a special transition that is related to dynamic tasks (discussed in the next section).

3- Tasks relations

Relations between tasks control the task execution sequence towards attaining the goal of the parent task. Relations take the form of ECA (Event, Condition and Action) rules:

Event	ON TS.State	TS is the source task
Condition	TD.State= "value"	TD is the destination task
Action	TD.Transition	

An example on relations is:

```
ON Fill_Flight_Search_Info.Completed
        If (Display_Flight.State="Created")
                Display_Flight.offer
```

Because transitions are un-ambiguously defined in the task's state diagram, the condition part can be automatically verified when executed in the condition. Thus, we can omit the condition part from the relation.

The action part is ensured to be executed in the relation, but the destination task may not change its state. Changing the destination task state depends on the transition condition. If the transition's condition is evaluated to true, the state is changed.

Task relations can be seen as adding an "AND" condition (the event and condition) to the task's transition defined in action part. In the example above, the task Display_Flights changes its state to "Started" if the offer transition condition is evaluated to true and the Fill_Flight_Search_Info state is "Completed".

Relations can be defined on sibling tasks in the hierarchy. Children tasks can also be related to direct parents. An example on defining relations between tasks is depicted in figure 4.4. In this figure, we note that a child task has a relation with the parent to complete it upon completion.

4- Optional tasks

Optional tasks are tasks that always look to parent tasks as completed, although their state diagram indicates a different state. An optional task has the property "isOptional" set to true.

5- Task Repetition

Repetition can be implemented on a task T using relations like: On

T.Completed, T.offer. Anyway, this cannot be executed at run time, because the "offer" transition is not possible from the "Completed" state. The rule should be updated to: On T.Completed, T.create. The latter rule means a new instance of the task T is created once the task T is completed. We call the newly created task instance a dynamic task.

A dynamic task is a task that is created by another task at run-time. Creation of these tasks (repetition decision) could be initiated by an explicit action from the user or by the system. The need for a decision for repetition justifies the need for a different task (the justification for identifying this task is a change of time::decision point). Thus, to repeat a task "T", we need a task for the decision "R". The relation between these tasks to enable repetition can be defined like: On R.Completed, T.create. R is called the pumping task (it pumps dynamic tasks in the model).

If the repetition decision is taken by the user, the pumping task must be interactive and dynamic. If the repetition decision is taken by the system, the pumping task needs to have the property "repeatable" set to "true". This is important because it has implications on the state diagram: an automatic transition is defined from the completed state to the created state. The repetition condition in a system-pumping task is the condition of its offer transition.

5.3 Implementation

We implement all linguistic levels using Java. All models are implemented as a programming language we call: The Prism Programming Language. We do not have a graphical designing tool for our linguistic tools for the moment. We use the Antlr[35] tool to parse our Prism programming language.

5.3.1 The Prism programing language and interpreter

A Prism program is a text file that contains statements to execute on a level. The Prism program is parsed and executed by an interpreter that translates these statements into a GUI at the targeted level.

Implementing linguistic models using an interpreter is important to satisfy the linguistic modelling requirement: "Levels shall be treated at run-

[35] www.antlr.org

time". Any level can be modified at run-time by statements that are interpreted by the interpreter. The interpreter has the simple UI to execute Prism programs depicted in figure 5.3.

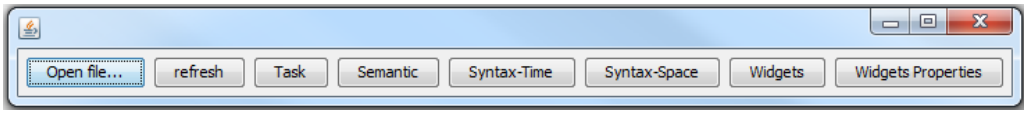

Figure 5.3 The Prism interpreter GUI.

The "open file" button in the interpreter GUI opens a prism program file. Other buttons render this file on the denoted level.

A Prism program contains different statements categories per-level. A statement has a prefix to denote the level to execute on. Anyway, a file can contain one category of statements or several. This allows extending a program per-level. Figure 5.4 depicts the structure of a Prism program.

```
prism{
    task:: statement;  //a statement to execute on the task level
    sem:: statement;   //a statement to execute on the semantic level
    synt:: statement;  //a statement to execute on the syntax-time level
    syns:: statement;  //a statement to execute on the syntax-space level
    lex:: statement;   //a statement to execute on the widget (lexical) level
    alpha:: statement; //a statement to execute on the widget properties (alphabetical) level
}
```

Figure 5.4 The structure of a Prism program.

In the implementation section for each level, we will explain the statements used at that level. For now, we present the BNF grammar-rules to parse the task level.

5.3.2 Prism task statements

The task model has four statements: define a task, remove a task, define a relation and finally remove a relation.

1- The define task statement

The grammar rule for the statement to define a task is:

```
taskStatement: id=Identifier '<' taskPropertyList '>'
```

The "`taskPropertyList`" sets attributes of the task. An example on defining a task is:

```
task:: searchFlight<name="Search for a Flight",
type=abstract, canRollback=stateless, canCancel=true, repeat-
```

```
able=false >;
```

This example defines the task "Search for a Flight", with "searchFlight" as the unique id for it. It also sets values for different task attributes, like: type, canRollback, canCancel and repeatable.

2- The define relation statement

The grammar rule for the statement to define a relation is:

```
taskStatement:'on' condition=expression 'action' ac-
                tion=expression
```

An example on defining a relation (an ECA rule) is:

```
task:: on display.completed action searchFlight.complete();
```

The example defines an ECA from the task display to the task search-Flight. Completing the task display completes the task searchFlight.

3- The remove task statement

The statement to remove a task is:

```
taskStatement:'remove' id=Identifier
```

An example is:

```
task:: remove display;
```

4- The remove relation statement

The statement to remove a relation is:

```
taskStatement:'remove' 'action'  action=expression
```

An example is:

```
task:: remove action display.complete();
```

5.4 Validation

5.4.1 External validation

We validate the task model on the case study "Search for a flight". The scenario is: *The user fills in search parameters. The system searches for relevant flights and displays them on the screen. The user selects the preferred task.*

1- The task model

The case study has the goal to "find a desired flight". This goal is decomposed into the following tasks with justifications and configurations in table 5.2.

Id	Parent	Task	Justification	Configuration
1	/	Search for a flight	Grouping	Abstract, Stateless rollback,
2	1	Fill Search params	-	Interactive, State-full rollback, canCancel=true
3	1	Display Flights	Grouping	Abstract, Stateless rollback,
4	3	Repeat on flights	Repetition.	System, Stateless rollback, repetitionTask=true
5	3	Display a Flight	From 2: Change in nature: interactive->system	System, Stateless rollback, canCancel=false
6	3	Select the Flight	From 4: Decision point	Interactive, Stateless rollback, canCancel=false

Table 5.2 Decomposition of the goal "Search for a flight".

The task "Display Flights" is a repetitive task, because for each flight the system displays, the user has the choice to select it or select another one. Thus, the notation enforces adding a system-pumping task that pumps the tasks "Display a Flight" and "Select the flight". That pumping task repetition condition is the number of flights in the search result.

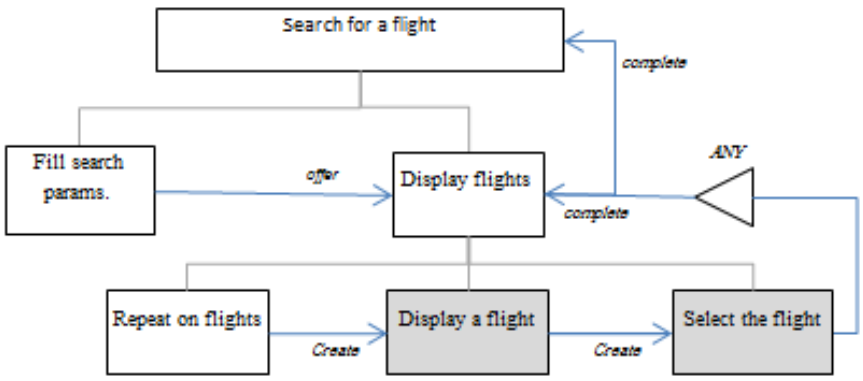

Figure 5.5 The "Search for a flight" case study task model.

From a linguistic perspective, Figure 5.5 provides the means to identify task input elements. Each dynamic interactive task "select the flight" requires a task input element for completion. The total number of identified task input elements at run-time is equivalent to the number of flights in the search result. Besides, other task input elements can be identified for each task from its state diagram and transitions. Every "User" transition requires a task input element to enable the transition.

2- The task GUI

We implemented the case study using the prism language and the prism interpreter. The prism program at the task level is depicted in figure 5.5.

```
prism{

    task:: searchFlight<name="Search for a Flight", type=abstract, canRollback=stateless, canCancel=true,
    repeatable=false >;
    task:: fill<name="Fill search params.", type=interactive, canRollback=stateful, canCancel=true,
    repeatable=false, parent=searchFlight >;
    task:: display<name="Display Flights", type=abstract, canRollback=stateless, canCancel=true,
    repeatable=false, parent=searchFlight >;
    task:: on fill.completed action display.offer();
    task:: on display.completed action searchFlight.complete();

    task:: getFlights<name="Repeat on flights", type=system, canRollback=stateless, canCancel=true,
    repeatable=true, parent=display >;
    task:: displayRow<name="Display a flight", type=system, canRollback=stateless, canCancel=true,
    repeatable=false, parent=display >;
    task:: selectRow<name="Select the flight", type=interactive, canRollback=stateless, canCancel=false,
    repeatable=false, parent=display >;

    task:: on getFlights.completed action displayRow.create();
    task:: on displayRow.completed action selectRow.create();
    task:: on any selectRow.completed action display.complete();
}
```

Figure 5.6 Implementation of the case study in the Prism language.

Executing this program in the Prism interpreter, we get the GUI in figure 5.7.a. There are several things to note in the figure:

1- Every task is rendered inside a rectangle. Children tasks are rendered as rectangles inside the parent rectangle.
2- The rectangle Colour reflects the current state of the task. Colours mapping to states is explained in the figure.
3- Produced task input elements depend on the configurations set on the task. Notice the appearance of the "cancel" task input element on the task "Fill search params." While it is not generated in other tasks.
4- The repeatable task "Repeat on flight" did not start pumping dynamic tasks because the condition is not satisfied yet.

Pressing the "complete" button in figure 5.7a changes the states of tasks and produces the GUI in 5.7b. In this GUI, we notice how the repeatable task is repeated 2 times and resulted in the generation of four dynamic tasks in total. The number of repetitions depends on the number of flights returned from the previous task. This can be determined at the semantic level. The interpreter repeats dynamic tasks 2 times by default if no repetition condition is set.

Selecting a flight can be performed by clicking the corresponding "complete" button in figure 5.7b. Doing so generates the GUI in figure 5.7c. In this figure we notice that the root task is completed and we have now a flight selected. Anyway, there is a possibility for the user to change her/his mind and change the selection of the task by clicking on the rollback "button". The user also has the possibility to conduct a new search by clicking the rollback button on the "Fill search params" task.

1- Comparison with other notations

We compare the notation used in our linguistic task model with that of CTT, as the mostly known notation for task models. This comparison is illustrated in figure 5.8. This figure shows how temporal relations can be represented using our notation. The linguistic task model notation encompasses the CTT one.

5.4.2 Internal validation

We elicited linguistic modelling requirements in table 4.1. The linguistic task model is concerned with four of them: one for all levels, in addition to three for task level.

Our linguistic task model satisfied all of the four linguistic modelling requirements. We discuss how that is achieved below.

1) **Levels shall be treated at run-time:** Satisfied by implementing an interpreter that can interpret prism statements. Interpretation is made at run-time, therefore, the task model is designed and manipulated at run-time.
2) **Task input elements are exclusively eligible to change a task state:** task input elements are calculated from the task state diagram. They cannot be created manually, and have a predefined behaviour: change the task state. The task state is protected against modifications by lower-level input elements.

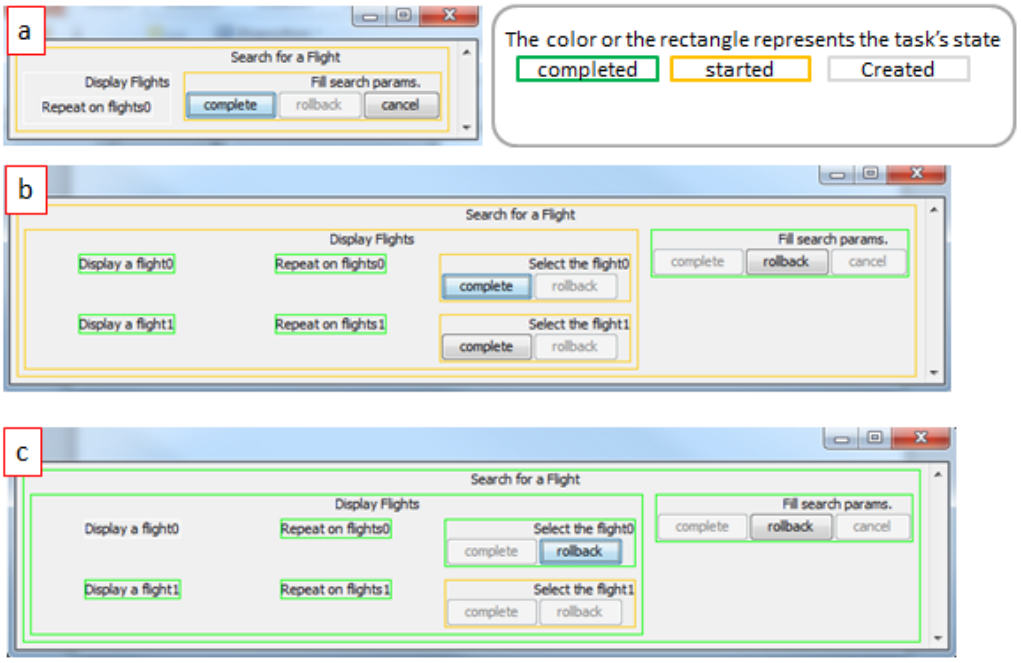

Figure 5.7 The task GUI of the case study.

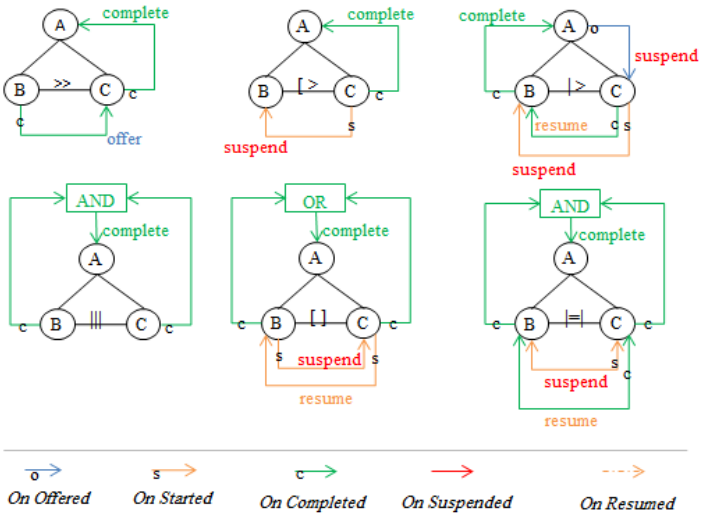

Figure 5.8 CTT temporal relations using the linguistic task model notation.

3) **Exclusively allow un-defining/enabling/disabling task UI elements:** All properties of task input elements are protected from modifications on lower levels by the interpreter.
4) **Task UI elements may not be refined to visual elements on UI:** The task model avoids any UI design-related aspects. It produces task input elements with no constraints on later design options.

5.5 Summary

In this chapter, we introduced a linguistic task model, implemented and tested it.

We validated the model on a simple case study, and assessed internal validation by showing how it fulfils linguistic modelling requirements.

The proposed linguistic task model is a proof of concept. In order to turn it into a real development tool, supporting tools should be implemented. A graphical task designer is one of the most important tools to implement. Besides, more external validation testing should be conducted.

In the context of this thesis, it is enough to have a proof of concept on the perspective and the linguistic modelling approach. Future work can focus more on specific properties and benefits of the perspective.

The task-UI is generated at the task level. It is far from being final (5 levels below). In the next chapter, we will see how the semantic model revives the task UI by mapping to the data model and making a working UI, although not final.

Chapter 5. A GUI Linguistic Model: The Task Level

Chapter 6 A GUI Linguistic Model: The Semantic Level

6.1 Context

In this chapter, we present the linguistic model on the semantic level. This model defines new concepts and realizes the ones coming from the task level. We also implement and validate it.

External validation of the model is applied on the same case study as presented in the previous chapter. Internal validation demonstrates how linguistic modelling requirements on the semantic level are satisfied.

6.2 The Semantic model

The semantic model is mainly concerned with:

1) Interfacing with the task model: by realizing what the task model is expecting from it.
2) Interfacing with the data model: the data model is part of the system design, not the GUI. Anyway, it impacts the GUI. This impact is the concern of the semantic model.
3) Flow of data in the UI: the UI may gather data from the user in a different way than presented in the data model. The flow and exchange of data among UI elements are controlled at the semantic level.
4) Defining needed UI elements: how many input/output elements do we need to carry out a task? What types of UI elements are needed?

This section addresses all the above functionalities to build a linguistic semantic model.

6.2.1 Interfacing requirements from the task level

The linguistic semantic model refines the tasks defined on the task level. It should produce all input and output elements that are needed to carry out a task. Figure 6.1 denotes an overview of the relations among the task model, the semantic model and the data model. The refinement of the task model takes the form of requirements for the semantic level: what the task model is expecting from the semantic model to implement.

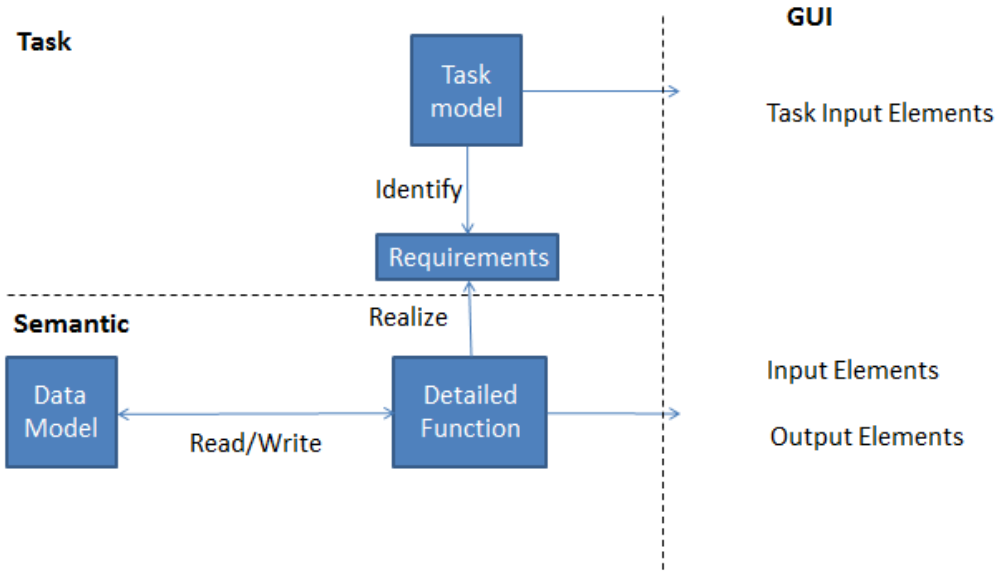

Figure 6.1 Interfacing between the task and semantic models

Each task is realized (refined) using one detailed function that should fulfil interfacing requirements for that task. Requirements of a task depend on the type of the task and the configured state diagram.

- System task: **must** have a detailed function to refine it. We need at least to realize the condition to complete. Realization of other state changes is optional.
- User/Mechanic task: **does not** need to be refined.
- Interactive task: A detailed function **might** be required to carry out the task. The detailed function can produce needed UI elements to

carry out the task.

From the above, we conclude that there are three types of interfacing requirements from the task level:

1) Obligatory requirements: a system task must be realized by a detailed function. An error is issued if such requirement is not fulfilled.
2) Tentative requirement: it is likely important to realize this requirement. An example is realization of an interactive task state change.
3) Optional requirement: it is likely unnecessary to realize this requirement.

6.2.2 An overview of the semantic model

Figure 6.2 depicts the general structure of the semantic model. Note that the outcome of this level to the GUI is the set of input and output elements needed to carry out the task. They sum up to the already-identified task input elements from the above level to produce the GUI at the semantic level.

The user may require different information from the task depending on its state. If the task is started, the user may need input elements to fill data. If the task is destroyed, the user does not expect any kind of interaction from the task. We define a **visible task state** as a state that may require interaction from with the user. Visible task states are: started, completed, suspended and errored.

In the figure 6.2 we notice that UI elements are categorized by facet. We define 4 facets based on visible states in the task model:

1- **Started facet**: What UI elements should be produce when the task is started? UI Elements in this facet are required to carry out the task.
2- **Completed facet**: What output elements should be produced when the task is completed? Eventually, no input elements are allowed.
3- **Suspended facet**: what output elements should be produced when the task is suspended? Eventually, no semantic input ele-

ments are allowed. Resuming a suspended task is defined in the task model.

4- **Errored facet**: what output elements should be produced when the task is errored? Eventually, no semantic input elements are allowed.

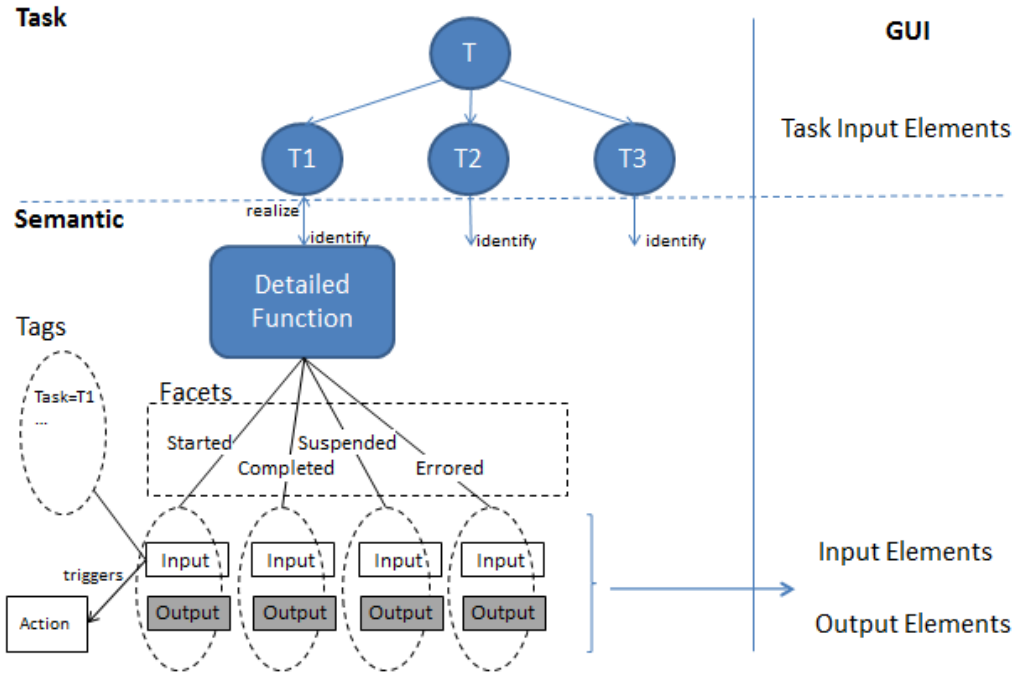

Figure 6.2 An overview of the semantic model.

Each input element requires an input action. An input action is a code snippet that is executed when the input is acquired. For instance, when the user clicks on a save button, the code that saves data is executed.

UI elements can be tagged by a *(key,value)* pair. This is a flexible way to implement logical relations among UI elements. The relation is depicted in the "key", and the role of the input element in this relation is depicted in the "value". For instance, each UI element in a detailed function is automatically tagged with the realized task. This is depicted in the figure: (task, T1). Manual tags can be also added as needed.

6.2.3 Data exchange

To realize a task model, we certainly need to exchange data among tasks. Data exchange is performed on the semantic model. This includes:

1- Exchange with the data model.
2- Exchange data inside the task: more precisely, inside the detailed function.
3- Exchange data between tasks: more precisely, between detailed functions.

Data exchange with the data model can be supported by exploiting the data model objects inside the detailed function. Exchanging data inside a task can be supported by defining local variables.

Exchange data between tasks is restricted by the hierarchy of tasks. A child task can access the context of the parent task to exchange data with. The only exception to the hierarchy is in dynamic tasks. When a sibling task creates a task, the created task can access the creator to exchange data with. Overall, data exchange between tasks is limited to the parent task and the creator. This is depicted in figure 6.3.

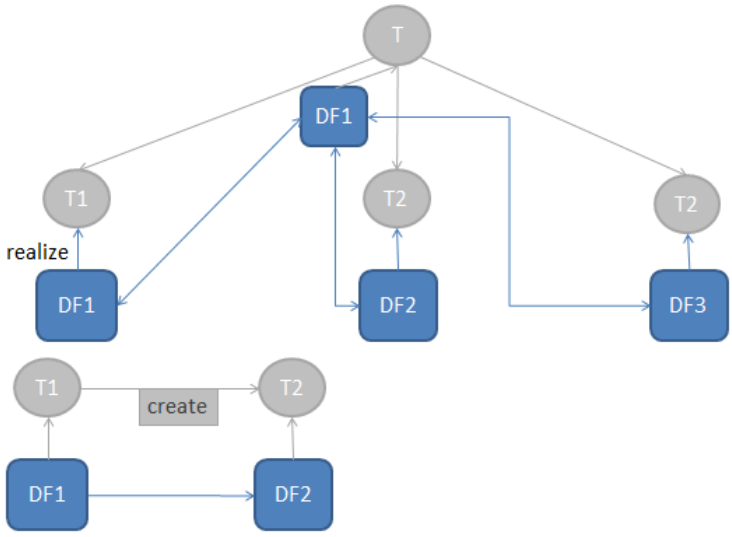

Figure 6.3 Data exchange between tasks in the semantic model.

6.2.4 Semantic GUI elements

Input elements are the way for the user to interact with the system. Interaction means exchange on information between the user and the machine. We define three types of input elements, based on who provides the exchanged interaction information, and the type of that information. These types are:

1) A User Data Input: this type of input elements allows the user to provide explicit information in the interaction. An example is an input element to fill in the user name. In this case, the provider of the exchanged information is the user.
2) A System Data Input: the system may provide the interaction information instead of the user. In this case, the user selects one or more items among choices provided by the system. This type of interaction requires a different type of input elements that we call: system data input elements.
3) A Processing Input: in the case when the system requires approval from the user on an action, there is no explicit data exchanged between the user and the system. In fact, there is an implicit agreement on what is changed, which is *the intention of the user*. In a GUI, when the user clicks on a button to approve an operation, the intention of the user (the approval) is transferred in the click itself. Therefore, the user is the provider of the information, but the exchanged information is implicit. A dedicated type of input is defined for this type of inputs, which is the processing input type.

These types of UI elements are important to modeling. A User data input requires a property to store the filled information. A System data input requires a read-only property that cannot be modified by the user or even at any lower linguistic level.

We also identified two types of output elements:

1) An Output Element: this is an element that contains information to display on the screen.
2) A Data Output Element: this is an output element that aggregates other output elements. It helps as a logical group of output elements when produced from a list of data, with no conceptual differences from output elements.

6.2.5 The semantic model UML diagram

Figure 6.4 depicts the semantic model UML diagram. It encompasses all the concepts discussed above. Notice that actions on UI elements are methods in the detailed function. Actions are modelled through the triggering relation between the InputElement and the Method classes.

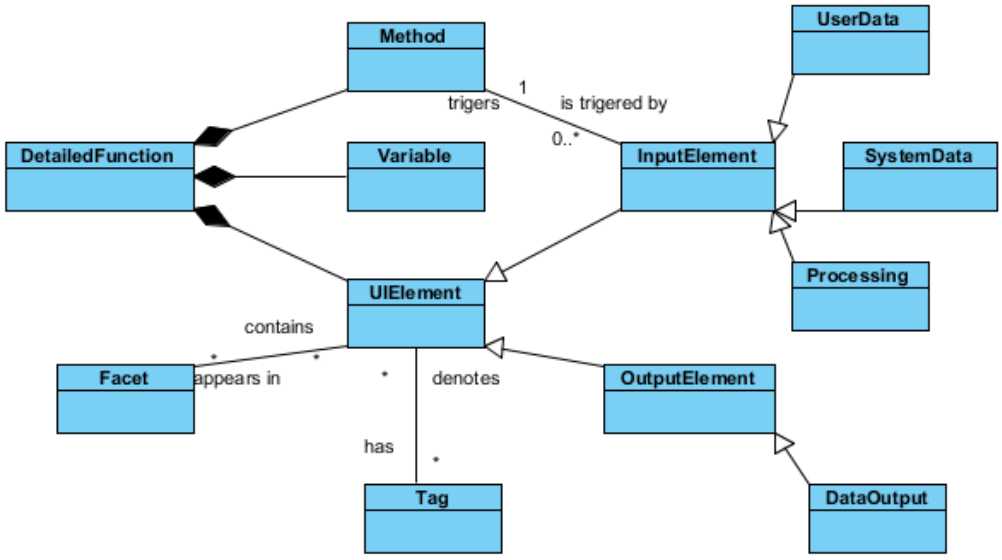

Figure 6.4 An overview UML diagram for the Semantic model.

6.3 Implementation

We continue to enrich our Prism programming language to support the semantic level.

As one can see from the semantic model UML diagram, the detailed function is a class that is similar to Java classes. Therefore, detailed functions should be implemented in Prism programming language as classes.

In order to minimize the implementation effort, we chose a pragmatic solution to implement detailed functions in Prism. Detailed functions are implemented in Java, and the Prism language supports importing Java classes.

We developed a Java class for detailed functions. To customize the de-

tailed function for the specific needs of a task, our class can be extended in Java and imported in the Prism program.

The definition of UI elements is supported in the Prism language. This would enable developing prototype of the GUI without the need to get into the data design and details.

6.3.1 Prism semantic statements

Prism supports two statements at the semantic level, which we explain in the following sections.

1- The realize task statement

This statement realizes a task from the upper level. The formal grammar rule for this statement is:

```
semanticStatement:'realize' task=Identifier 'as' 'java'
                  'type' type=StringLiteral
```

This statement allows importing a Java class that extends our Java detailed function class into prism, in order to realize a task. An example of using this statement is:

```
sem:: realize searchFlight as java type "seman-
        tic.v3.SearchForFlight";
```

Where, "searchFlight" is the id of a task defined at the task level, and "semantic.v3.SearchForFlight" is the Java class that realizes this task.

2- The define a UI element statement

This statement defines a UI element in a detailed function. The formal grammar rules for this statement are:

```
semanticStatement: 'def' uiElementType '(' default-
Value=StringLiteral ')' task=Identifier '.' uiEle-
meId=Identifier 'in' 'facets' '<' facetsList '>' ;

uiElementType: 'output' |'dataOutput' |'systemInput'
|'userInput'| 'processing';
facetsList: facetName (',' facetName)*;
facetName: 'STARTED'|'COMPLETED'|'SUSPENDED'|'ERRORED';
```

This statement allows defining a UI element with id "uiElemId" of a specific type (defined by uiElementType), in the detailed function that

realizes the task "task". It also determines the facets it belongs to. An example on using this statement is:

```
sem:: def output("Flight number") searchFlight.flightNo in
                    facets<COMPLETED>;
```

Where, "searchFlight" is the id of a task defined at the task level, and the defined UI element is "flightNo" of type "output" and a default displayed value is 'Flight number". This UI element will appear in the "completed" facet only.

6.4 Validation

6.4.1 External validation

Let's continue on the case study on the search for a flight we discussed earlier in the validation of the linguistic task model.

We have more information on how to carry out each task. The new information is mentioned below.

The "Fill search params." task allows the user to search by:

- From airport
- To airport
- On a specific date.

Each row resulting from the search on the database contains:

- The flight number: flightNo.
- The ticket price: flightPrice.
- The number of free seats in the flight: flightSeats.
- The airline company: flightCompany.

Besides, the user needs to be able to sort flights per price. It might help in making a decision. Finally, the user should be informed on the selected flight.

1- The semantic program in Prism language

In the Prism program for the task model, we add the following semantic statements to define UI elements and realize tasks by detailed functions. Figure 6.5 illustrates these statements.

```
sem:: realize searchFlight as java type "semantic.v3.SearchForFlight";
sem:: def output("") searchFlight.flightNo in facets<COMPLETED>;
sem:: def output("") searchFlight.flightPrice in facets<COMPLETED>;
sem:: def output("") searchFlight.flightSeats in facets<COMPLETED>;
sem:: def output("") searchFlight.flightCompany in facets<COMPLETED>;

sem:: realize fill as java type "semantic.v3.FillSearch";
sem:: def output("from") fill.from_label in facets<STARTED,COMPLETED>;
sem:: def userInput("") fill.from in facets<STARTED,COMPLETED>;
sem:: def output("to") fill.to_label in facets<STARTED,COMPLETED>;
sem:: def userInput("") fill.to in facets<STARTED,COMPLETED>;
sem:: def output("date") fill.date_label in facets<STARTED,COMPLETED>;
sem:: def userInput("") fill.date in facets<STARTED,COMPLETED>;

sem:: realize display as java type "semantic.v3.DisplayFlights";
sem:: def processing() display.sortByPrice in facets<STARTED>;

sem:: realize getFlights as java type "semantic.v3.GetFlights";

sem:: realize displayRow as java type "semantic.v3.DisplayRow";
sem:: def output("") displayRow.flightNo in facets<STARTED,COMPLETED>;
sem:: def output("") displayRow.flightPrice in facets<STARTED,COMPLETED>;
sem:: def output("") displayRow.flightSeats in facets<STARTED,COMPLETED>;
sem:: def output("") displayRow.flightCompany in facets<STARTED,COMPLETED>;

sem:: realize selectRow as java type "semantic.v3.SelectFlight";
```

Figure 6.5 The semantic level implementing of the case study in Prism language.

We also implemented Java classes for detailed functions as depicted in the Prism code. Data exchange and actions on input elements are handled inside these classes. They are normal Java classes and not of a big interest to the reader, se we do not present them.

2- The semantic GUI:

Executing the Prism program at the semantic level gives the GUI depicted in figure 6.6a. We see an enriched version of the GUI at the task level. The "Fill search params" Task is more refined by adding required UI elements. The user can now search for a flight.

Pressing the complete button produces the GUI in figure 6.6b. This GUI is also a refined version of the GUI at the task level. The GUI at the semantic level is a working GUI. It is mapped to the data model and can produce real results. The search we carried out resulted in 4 flights. The user can click on the complete button next to the flight to select it.

Another interesting aspect to note is how facets perform on the task

"Fill search params.". When the task was in the started state, we had 3 user input elements. These input elements were defined in two facets: started and completed (review the semantic prism code in figure 6.5). In the completed state, no input elements are allowed; therefore, the semantic model converts them into output elements.

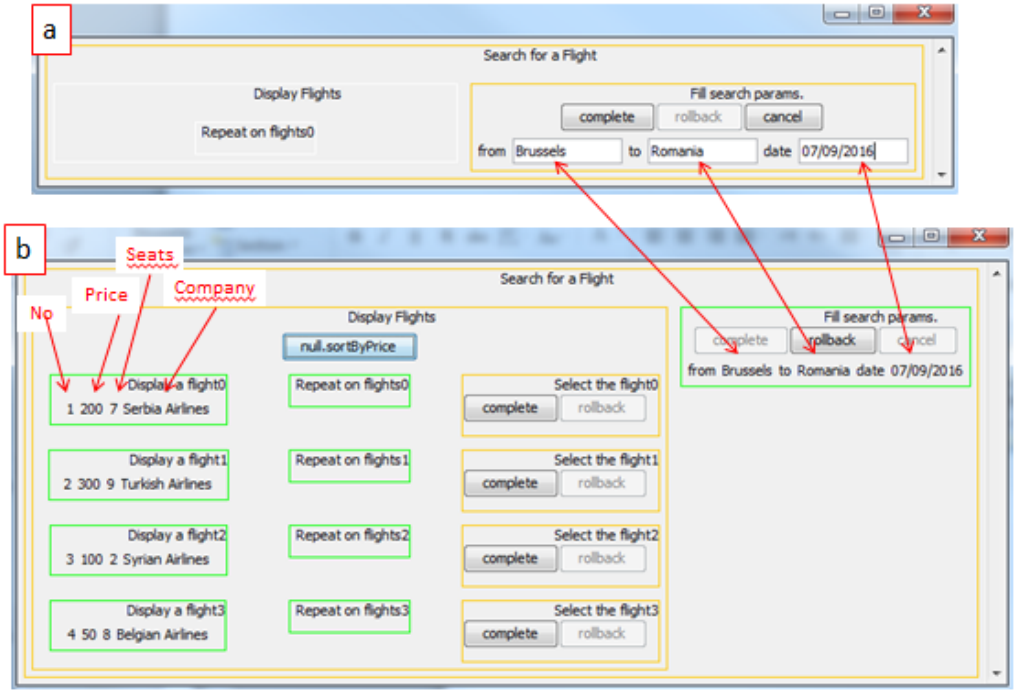

Figure 6.6 The GUI at the semantic level to search for a flight.

If we click on the "sortByPrice" button, flights get sorted per price. This is depicted in figure 6.7. Selecting a flight leads to the completion of the root task as we have seen in the task GUI. The difference from the task GUI is in the appearance of the information on the selected flight.

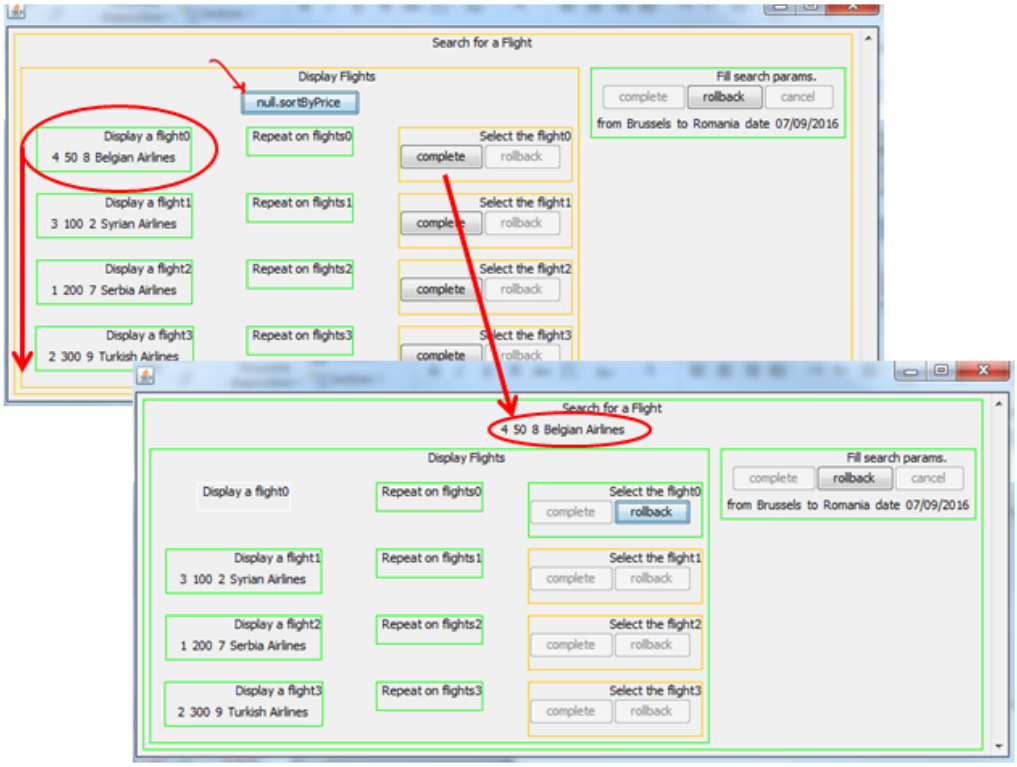

Figure 6.7 The GUI at the semantic level after selecting a flight.

The semantic GUI demonstrates the separation of concerns between the task and the semantic level. The flow of execution of tasks cannot be breached by the semantic level. The execution on the semantic level showed the same GUI we have already executed on the task level, but with more refinement. Each level is concerned with its own concepts and activities, which do not conflict with each other, thanks to the linguistic perspective.

The reader should not be deceived with the interpreter behaviour. The interpreter generates widgets based on a default mapping between input elements types and widgets. This can be refined later at the widgets level. Besides, the interpreter has also pre-defined placement rules, which will be defined later on the syntax-space. The behaviour of the interpreter is defined in order to provide an acceptable usability level of the GUI at different levels.

6.4.2 Internal validation

We elicited linguistic modelling requirements in table 4.1. The linguistic semantic model is concerned with four of them. We explain how these requirements are satisfied. Note that the satisfaction of the requirement on all levels is already discussed on the task level, so we will not get back to it.

1) **Semantic input elements are exclusively eligible to launch a detailed function:** this is guaranteed by the interpreter that does not expose detailed functions to other levels.
2) **Allow defining logical groups on UI elements:** Satisfied using tags on UI elements.
3) **Exclusively allow un-defining/enabling/disabling semantic UI elements:** All properties of task input elements are protected from modifications on lower levels by the interpreter. The task level does not know about them.
4) **Semantic UI elements may not be refined to visual elements on UI:** The semantic model avoids any UI design-related aspects. It produces UI elements with no constraints on later design options.

6.5 Summary

In this chapter, we introduced a linguistic semantic model, implemented and tested it.

We validated the model on a simple case study, and assessed the internal validation by showing how it fulfills linguistic modelling requirements at the semantic level.

We also presented how to produce a working GUI at the semantic level. We demonstrated how the refinement of the GUI is carried out by refining tasks defined at the task level on the semantic level.

The proposed linguistic semantic model is a proof of concept. In order to turn it into a real development tool, supporting tools should be implemented. The Prism language needs to implement its own statements to define detailed functions independently from the Java language. Tool support is also important to foster the design. Besides, more external validation testing should be conducted.

Chapter 6. A GUI Linguistic Model: The Semantic Level

Chapter 7 A GUI Linguistic Model: Perceptual Levels

7.1 Context

In this chapter, we present the linguistic model on the syntax-time level. All needed UI elements to carry out a task are defined at the two upper levels. This model defines navigation among them. We also implement and validate the model.

External validation of the model is applied on the same case study in the previous chapter. Internal validation demonstrates how linguistic modelling requirements on the syntax-time level are satisfied.

7.2 Modeling the syntax-time level

This model has two main concepts:

1) Time containers: logical groups of elements that appear together at the same time.
2) Navigation elements: input elements that allow navigation from a time container to another.

The purpose of the syntax-time model is to define the navigation among time containers.

7.2.1 The syntax time model

1- Interfacing requirements

This model realizes the upper level by placing every UI element produced from upper levels in a time container.

Let's first define the time in the syntax-time model. The GUI moves from a state on the screen to another. This movement is directed by inputs: when an input is acquired, the GUI moves to another state. The time ticks/moments are inputs on the GUI.

A valid navigation model is: a model that permits accessing any active (enabled) input element at any moment. In other words, assume the semantic level produces *n* input elements (including task input elements). The user should be able to interact with the system using any of these inputs at that moment. An input element is a capability for the user. Therefore, a valid navigation model should not hide any of these capabilities from the user. Every input element should be accessible directly or indirectly.

2- A navigation notation

Assume we have two containers A and B. The navigation relation from one to the other can be defined as:

1) B is observable from A: both A and B are displayed (visible or active) at the same moment.
2) B is browsable from A: Only A is displayed, with the possibility to navigate to B.

Browsable navigation has different sub-types depending on the relation with A:

1) Modal: When B is displayed, A is hidden. See figure 7.1b. A contains an output element to display "Hello world", while B contains an output element to display "Good bye sailor".
2) Modal-partial: when B is displayed, A is partially hidden. See figure 7.1c and notice the difference from 7.1b in the definition of A. A contains an output element for "Hello world" and three input elements for "First", "Second", "Third" and "close". When B is displayed, only the output element for "Hello world" is hidden from A.
3) Inline: When B is displayed, A is also displayed. See figure 7.1a.

Note that browsable containers require a navigation input element to display them. Defining a browsable relation between two containers creates a navigation input element from one to another. We call this navigation element as: the opener of the container. Besides, a close input ele-

ment might be required for the browsed container to hide it, we call it: the closer of the container.

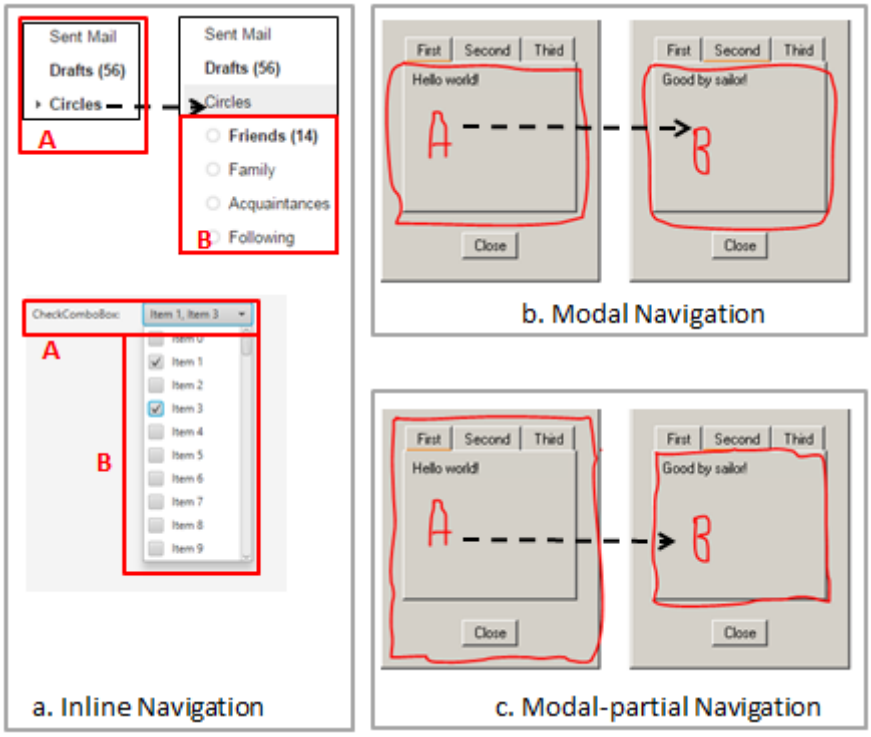

Figure 7.1 Examples on different types of browsable navigation between two containers.

To simplify discussing navigation among container, we created a notation to model navigation relations among containers. The notation is depicted in Figure 7.2.

The notation introduces the notation of a view axis. To model modal relations, we consider the user is looking at the container from a view axis. To display B alone, while hiding A, the user looks from a different view axis. Therefore, the navigation element to open a modal container B from a container A will simply change the view axis and A will no more be visible.

The notation is read as follows:

At a moment *t*, the GUI displays visible containers on the current view axis. The root container is visible at the beginning. A container is visible if:

1) It is observable from a visible parent.
2) It was explicitly set visible by an opener.

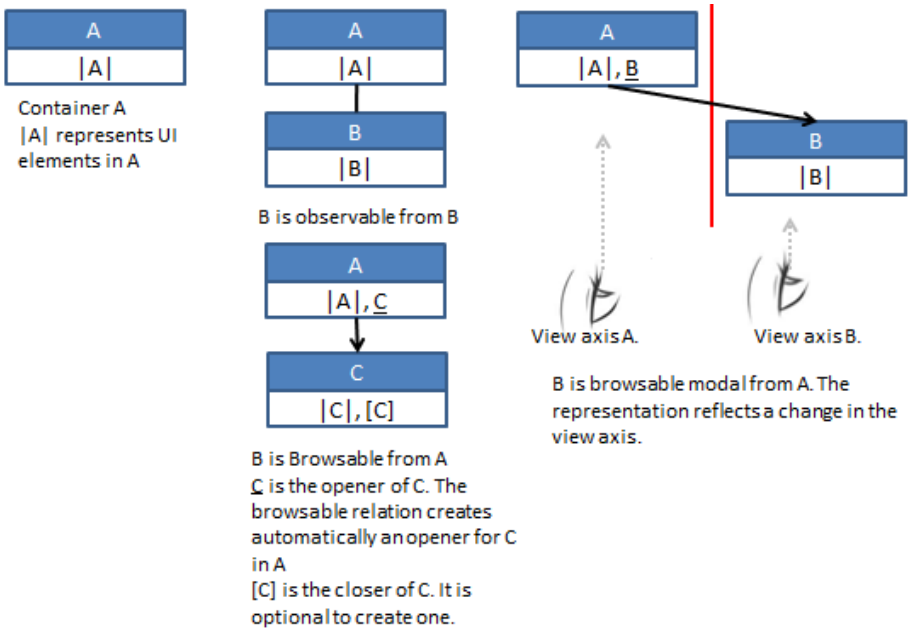

Figure 7.2 The navigation notation

A container can be explicitly hidden by its closed, if one is defined.

We still need to support the modal-partial navigation in the notation. We explain this in examples on using the notation.

3- Examples on using the notation

Figure 7.3a depicts the model for a drop-down box. Without the closer [b], once opened, it cannot be closed again. Note that the elements of B |B| are input elements grouped together in the same container. Compare the drop-down box model to the list-box model in 7.3b to note differences.

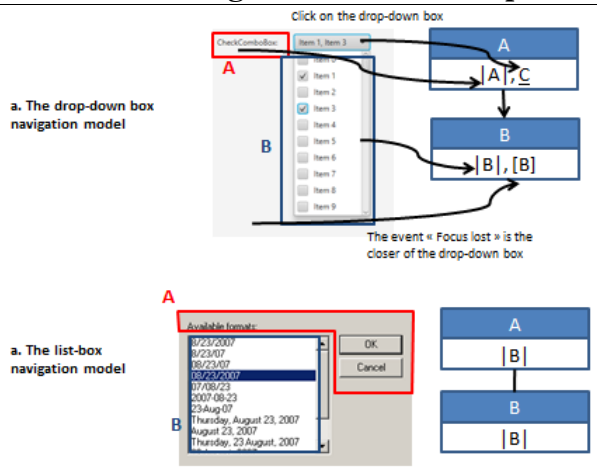

Figure 7.3 Modelling the navigation in a drop-down box and a list-box.

Figure 7.4 depicts a model for a GUI with grid with multiple pages. The model introduces a new notation: the view. A view is a way to display elements from a container on a view axe. Let's first explain how the grid example works to understand views better.

The relation of pages to the grid is modal. Each page creates its own view to exclude other pages. This exclusion affects the grid and its parents. It shows only a page and hides the complete GUI.

To solve this issue, parents of the grid should be visible on all view axes of its pages. Views can help in this issue by retaining elements from parents into the view axis they belong to. Therefore, we can create a view on each view axe to display elements from parents. This is a lot of work to solve this minor issue.

The browsable modal relation has two issues: (1) define the container it is browsable from, and (2) define the container it is modal on. A page is browsable from the grid container and modal on the grid also. It is not modal on A. Therefore, if we change the notation of the browsable modal relation to denote both related containers, we get rid of the problem of manually creating views for all parents. This new notation is more systematic and permits a program to create views automatically. The new notation is depicted in figure 7.4 as the enhanced notation.

Views are beneficial to modal-partial relations. A modal container B on the container A can define a view on A that retains part of the elements

of A. A view has a condition on elements it can retrieve from the targeted container.

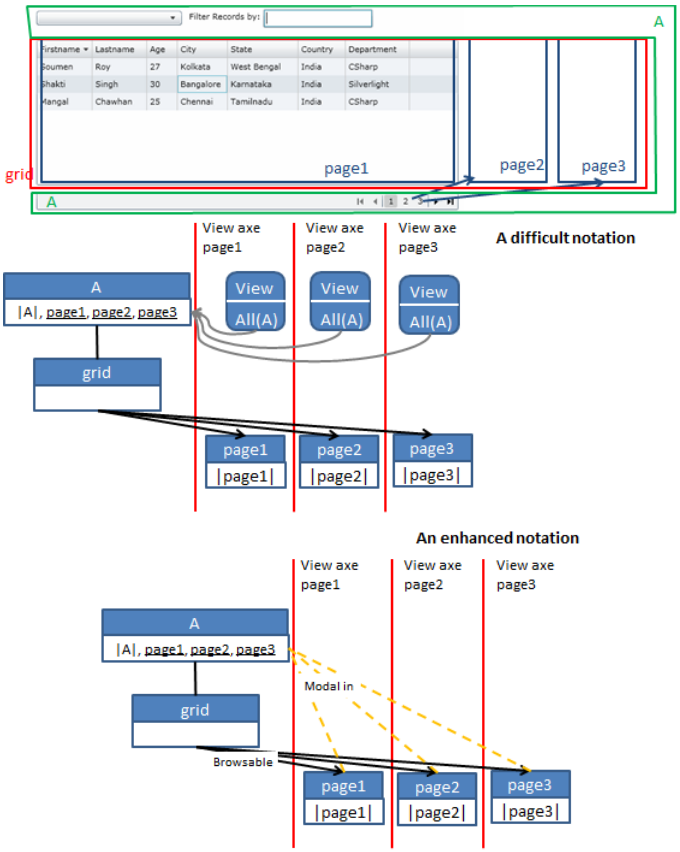

Figure 7.4 Modelling the pagination in a grid and the extended notation.

7.2.2 Implementation

We continue to enrich our Prism programming language to support the syntax-time level. We introduce new statements based on the notation introduced previously.

We define 6 statements that are categorised into 4 types:

1) Define a time container (one statement).
2) Define a relation (three statements).
3) Populate a time container with UI elements (one statement).

4) Explicitly create navigation elements on a container (opener and closer) (one statement).

1- The statement to define a time container

This statement defines a time container. The formal grammar rule is:

```
syntaxTimeStatement:'TimeContainer' c=Identifier ('['
                    exp=expression ']')?
```

This statement allows creating a time container or an array of them. It can be used as:

```
synt::TimeContainer grid;
synt::TimeContainer page[3];
```

2- The statement to define navigation relations

Three statements are built to define a relation. The formal grammar rules are:

```
syntaxTimeStatement:'observe' child=expression 'from' par-
ent=expression
 |'browse' child=expression 'from' parent= expression 'in-
line'
 |'browse'  container= expression 'from' parent = expression
'modal' 'on' ancestor= expression;
```

These statements allow creating relations in figure 7.4 as:

```
synt::observe grid from A;
synt::browse page[0] from grid modal on grid;
synt::browse page[0] from grid modal on grid;
synt::browse page[0] from grid modal on grid;
```

3- The statement to populate time containers with UI elements

We chose to use relational algebra to manage the way we populate time containers with UI elements.

If you look at UI elements as tuples, with their properties and tags as the attributes of the tuple, we can query these elements using relational algebra to populate time containers. Therefore, statements presented here are similar to those in the SQL programming language that is used to query a database. In our case, we have only one table that contains UI elements coming from the semantic level in addition to those generated

by browsable relations.

The formal grammar rule for the statement that populates time containers is:

```
syntaxTimeStatement:'select' e=Identifier 'from' list=qualifiedName 'where' condition=expression block=syntBlock;
```

This is a powerful statement that can query any UI element based on any attribute or tag. Examples are presented below.

This statement defines the modal relations between a grid and pages:

```
synt::select e from page where true {browse e from grid modal on grid;};
```

This statement moves the opener of page[0] in the A time container:

```
synt::select e from uiElements where e.timeContainer==page[0].opener { e.timeContainer = A; };
```

Where:

- uiElements is the list that container all UI elements.
- timeContainer: every UI elements has an pointer to the container time container.
- opener: the opener element of pages[0].

4- The statement to populate time containers with UI elements

This statement can be used to create closer elements because they are optional and cannot be deduced from relations. Besides, it is interesting to create openers in some cases we want to consider an already existing input element as the opener of a container. An example is to open the container that displays search results when the search button is pressed (the search button is a task button).

The formal grammar rule for this statement is:

```
syntaxTimeStatement:'open' container=qualifiedName 'on' uiElement=expression

|'close' container=expression ('on' uiElement=expression)?
```

This statement allows creating a closer element on a container like:

```
synt::close A;
```

It also allows overloading a task input element called "search.complete" on a search task to allow opening the results container, called "result" as:

```
synt::open result on fill.complete;
```

7.2.3 Validation

1- External validation

Let's continue on the case study on the search for a flight we discussed earlier in the validation of the linguistic task model.

The prism interpreter creates a default model on the syntax time level, based on the task hierarchy. It creates a time container for every task with observable relation between a child and its parent. UI elements produced from a task and its detailed function are automatically assigned to the related time container. The execution of the prism program without any syntax –time statement looks like in figure 7.5 (in debug mode to show containers names).

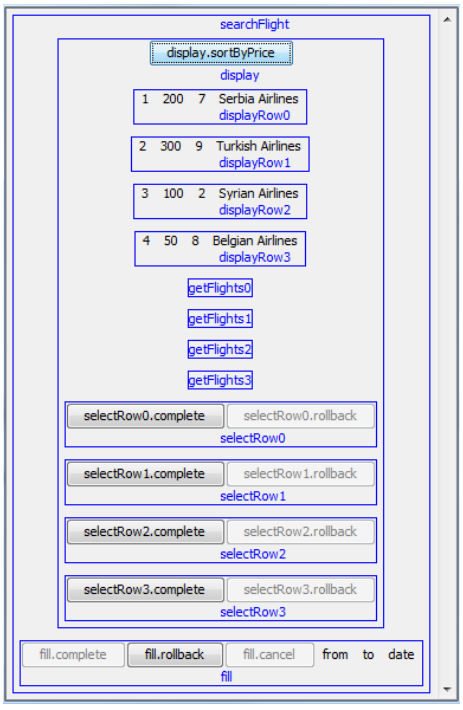

Figure 7.5 The initial search for flight GUI at the syntax-time level in debug

mode

We note the following remarks on the navigation as defined by default:

1) The search results are displayed even before pressing the search button.
2) The grid displays all the rows returned from the search. We want to define navigation: 2 elements per page.

The first step is to change the relation between the "display" and "fill" containers to browsable. This would fix the first remark, but it will create another: we need two clicks to show the results: one to execute the search and the other to open the "display" container. Therefore, we will overload the "fill.complete" to open the "display" time container. The statements we need are:

```
synt:: browse display from searchFlight inline;
synt:: open display on fill.complete;//fill.complete is the ui element id
```

Now let's group elements per-row. This would allow for easier pagination. We can do this by defining a "gridRow" array container, that are observable from "display" and populate each row with elements from "selectRow" and "displayRow". The statements to do this are listed below.

```
synt:: TimeContainer gridRows[display.getFlights.length];//Group elements observables per row in dispaly
synt:: select e from gridRows where true { observe  e from display;};
synt:: select e from uiElementsList where e.tags.index>=0 && (e.tags.task=="selectRow" || e.tags.task==
"displayRow")    { e.timeContainer= gridRows[e.tags.index]; };//populates each rows

synt:: TimeContainer gridRoot; //Define a container for the grid in display. Important to manage modal
```

Now the UI looks like in figure 7.6a. We show only the interesting part of it.

To create grid with pagination, we do as we have seen previously. The code is:

```
synt:: TimeContainer gridRoot; //Define a container for the grid in display. Important to manage modal
pages.
synt:: observe gridRoot from searchFlight.display;

synt:: TimeContainer pages[gridRows.length/2];//the pages. 2 rows per page.
synt:: select e from pages where true { browse e from searchFlight.gridRoot modal on
searchFlight.gridRoot;};//pages are modal in the grid
synt:: select e from gridRoot.uiElements where true  { e.timeContainer=searchFlight.display;};//Move
openers of grid children to display
synt:: select e from gridRows where true { observe e from pages[e.index/2];};//distribute gridRows on
the pages.
```

The final GUI at the semantic level is depicted in figure 7.6b, in produc-

tion model. Notice how paging on the grid is defined.

a.

b.

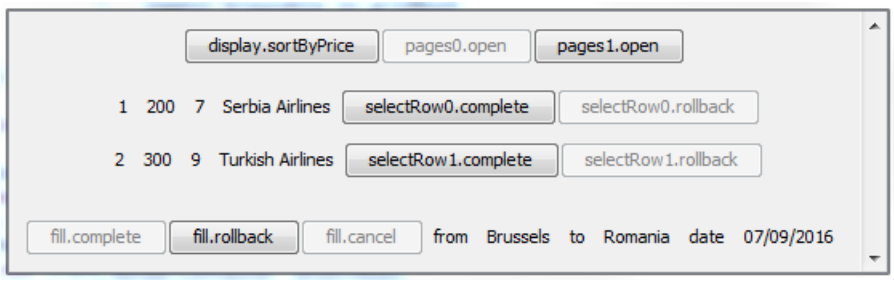

Figure 7.6 The GUI at the syntax-time level for the case study.

The most important thing to note in the case study is that whatever we do on this level cannot affect what is already done on the upper level. All functionalities from upper levels are still working here. Even changing the navigation model did not affect the functionality or the capability because a valid navigation model cannot hide any input element and should make it accessible to the user.

Another thing to note is this simplicity of the model at this level. This is because we are focusing on a specific issue in the GUI development.

2- Internal validation

We elicited linguistic modelling requirements in table 4.1. The linguistic syntax-time model is concerned with five of them. We explain how these requirements are satisfied. Note that the satisfaction of the requirement on all levels is already discussed on the task level, so we will not get back to it.

1) **Allow defining navigation elements:** Satisfied. The model automatically deduces containers openers. It allows defining openers and closers on any container, or even overloading an already-

existing input element to open/close a container.

2) **Navigation elements are limited to enable navigation from a time container to another**: Satisfied. Even overloading does not breach this requirement because the overloaded element becomes a navigation element in addition to original nature.
3) **Must place all UI elements from the semantic level, at run-time, on time-containers**: Satisfied thanks to the powerful statement to populate UI elements, based on any property they have, at run-time.
4) **Exclusively allow un-defining/enabling/disabling navigation elements:** Satisfied. This is guaranteed by the Prism interpreter that prohibits such activities.
5) Navigation elements may not be refined to visual elements on UI. The syntax-time model avoids any issues related to concretizing elements. It produces UI elements with no restrictions on later design options

7.2.4 Summary

In this section, we introduced a linguistic syntax-time model, implemented and tested it.

We validated the model on a simple case study, and assessed internal validation by showing how it fulfils linguistic modelling requirements.

We also presented a navigation model and notation. We demonstrated how to produce a working GUI on the syntax-time level. We also demonstrated how the refinement of the GUI is carried out by refining the navigation at this level.

The proposed linguistic syntax-time model is a proof of concept. In order to turn it into a real development tool, supporting tools should be implemented. For real life development, we need a designer tool to save the time in writing the Prism code. However, such a navigation designer tool can produce the Prism code that is interpreted and can be modified at run-time. Besides, more external validation testing should be conducted towards a real development tool.

7.3 Modeling the syntax-space level

This model has two main concepts:

1) Space containers: logical groups of elements to define placement of these elements on the screen
2) Displacement elements: input elements that allow navigation on the screen.

The purpose of the syntax-time model is to define placement rules among UI elements from upper levels.

7.3.1 The syntax space mode

1- Interfacing requirements

All UI elements should be placed in space containers. This is independent from the choice to widgets that concretize these UI elements. The simple principle is: if a UI element is produced, it should be placed somewhere.

A valid syntax space model is a model that does not prevent the interaction, by hiding UI elements behind each other, and preventing access to enabled ones. An example is an advertisement pop-up on top of a screen with no way to close it.

Note that syntax space rules cannot be applied before concretizing the elements, to know what is concretized as visible and what is not, in addition to know dimensions of visible widgets and their places (if set at the widgets properties level). We need to set general placement rules to guarantee a placement look and feel at this level.

For instance, in figure 7.6b, we need to ensure the search part appears above all elements. However rules on placing UI elements inside this part (like to display elements in a tabular form or a grid form) might introduce rigidity that restricts lower design options on how to physically place widgets on what exact positions (on the alphabetical level). The desired degree of rigidity can be defined at this level: the more rules among UI elements, the more rigid the design is. The linguistic syntax-space level can be a good place to impose usability guidelines on the placement of elements (as rules to respect by lower levels). This level can implement no rules, which gives all the freedom to lower levels to concretize the way they want and place elements where they want. It can also define rules among all elements and thus enforcing many restrictions on lower levels.

2- Placement rules

Assume we have two objects A and B. We can place these two objects on the screen using simple placement relations:

1) A above/below B.
2) A before/after B.
3) A on top/behind B.

These relations can define possible placements of B in relation to A. The figure 7.7 depicts the meaning of these relations. These relations are enough to define the general layout of the GUI at the syntax-space level, while giving the freedom to the widgets properties level to implement exact positioning with respect to these relations/rules.

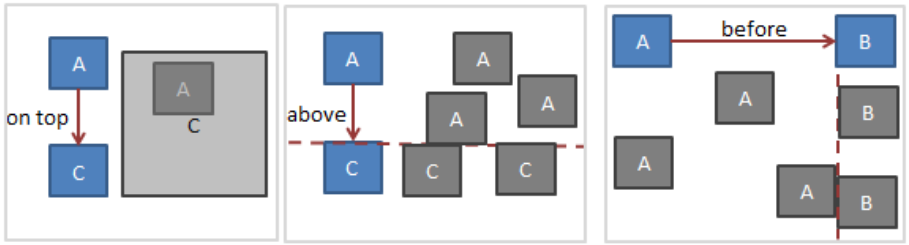

Figure 7.7 The placement relation and their meanings

3- Placement containers

A placement container is group of UI elements that organizes and facilitates defining relations among them.

Containers can be organised in a hierarchy. They are UI elements that might optionally be concretized as panel widgets. Only a leave space container can contain UI elements. An example on organizing placement containers, UI elements is depicted in the figure 7.8.

To ensure a valid placement model, space containers can define default placement rules among elements, if no rule is defined. Space container has a rulesHint property that can take one of two values: Horizontal and Vertical. A horizontal hint allows the placement container to add a before/after placement relation, if the container cannot determine their appropriate positions in a different way.

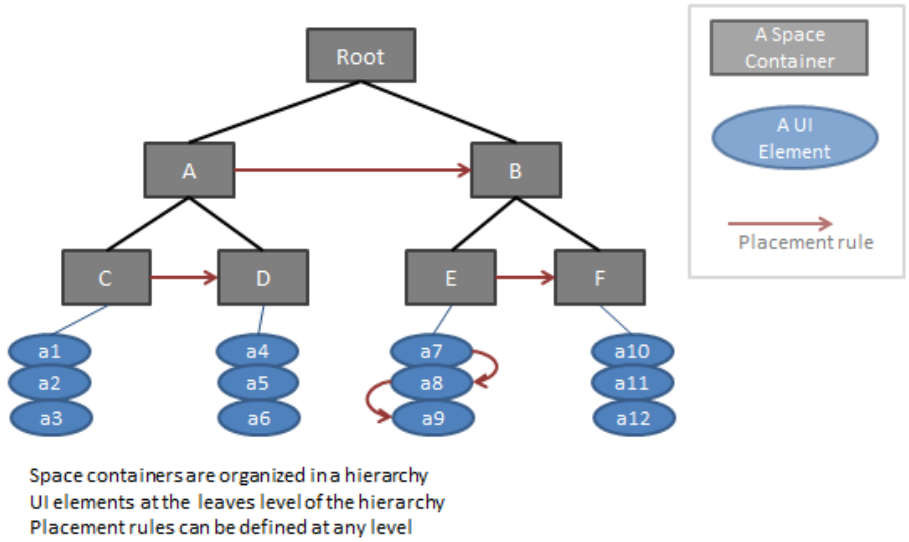

Figure 7.8 The hierarchy of space containers, UI elements and placement rules

4- The algorithm to execute placement rules

The syntax-space level is executed synchronously with lower levels. The execution algorithm is:

1) Syntax-space level: Create space containers and distribute elements on them
2) Widgets level: Concretize UI elements
3) Widgets properties level: Set attributes on concrete widgets
4) Widgets level: calculate exact dimensions and positions (if any is set at the lower level)
5) Syntax-space level: apply placement rules.

This algorithm ensures that when applying placement rules, we have all we need: Is the element visible or not, information on the dimensions of each widget, and information on the position of a widget (if set at the widgets properties level). To apply rules, we come through two steps:

1) Reduction: reduce the set of rules: by substituting rules to a subset that applies on visible widgets only.
2) Execution: execute the reduced set of rules to calculate/ensure placement of widgets on the screen.

The reduction of UI elements is straightforward:

1) If a widget concretizes several UI elements together, substitute these UI elements with the concretizing widget in all placement rules.
2) Invisible widgets are mapped to a visible widget (an event that applies a widget). Back to the first step.
3) Space containers that are not concretized redefine their rules on their children, and remove themselves from the rules list.
4) At the end, remove all relations from and to the same widget (relations to self are always satisfied).

The reduced set of rules becomes a set of rules on widgets. They can be executed to calculate dimensions and positions of widgets on the screen, or to ensure that their placement information are acceptable.

5- View Ports

A container calculates/validates the placement of its children on the screen. Depending on placements rules and on concrete placement information given to the widget at lower levels, its dimensions are unpredictable. On a GUI, we have a limited space and we need to display all widgets on the available surface. Therefore, we need to enable scrolling on space containers if their size exceeds limits.

A viewport is a visible area on the screen that is used to display a container inside. Figure 7.9 depicts a screen (the eclipse IDE main screen) that is divided into different viewports. Each viewport is attached to an underlying space container. The content of that space container is displayed with respect to the available area defined in the viewport.

A viewport has width and length constraints that should be respected. When the size of the underlying container exceeds these constraints, the viewport can define displacement input elements (scrolling). The viewport have the following attributes:

- WidthCalc/HeightCalc: can have one of four values:
    - FitContent: the viewport width is adapted to that of the children.
    - Available: the width represents a percent of the available space in the parent (0-100%) viewport.
    - Fixed: the width is a constant value.
- WidthMax/WidthMin/HeightMax/HeightMin: max and min values for width/height not to exceed.

- AllowScrolling: acceptable values are: Horizontal, Vertical, Both, None or Exception. When the content exceeds the max constraint, enable scrolling according to the value. If this value is exception, raise an exception to the interpreter.

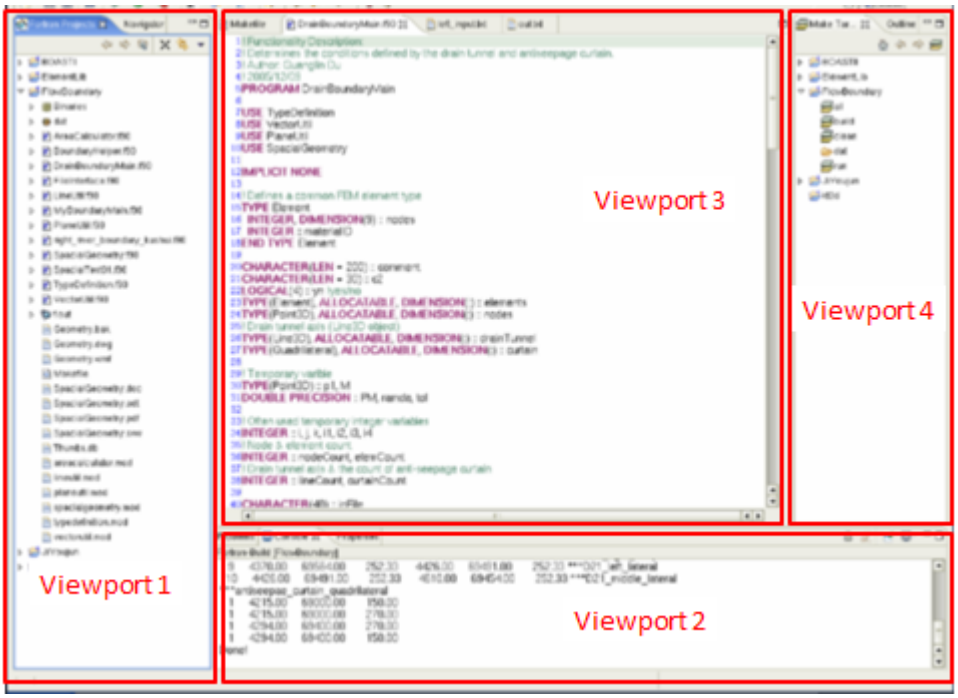

Figure 7.9 Viewports in a GUI. They provide scrolling to underlying containers.

7.3.2 Implementation

We continue to enrich our Prism programming language to support the syntax-time level. We introduce new statements based on the notation introduced previously.

These statements are not that different from on different levels. They are:

```
syntaxSpaceStatement:'SpaceContainer' id=Identifier ('['
exp=expression ']')? '<' (synsPropertyList)? '>'
#SynsDeclareContainerStat
|'ViewPort' id=Identifier 'on' container=Identifier  '<'
(synsPropertyList)? '>' #SynsDeclareViewPort
```

```
| 'select' e=Identifier 'from' list=qualifiedName 'where'
exp=expression block=synsBlock #SynsSelectorStat
|assignStatement #SynsAssignStat
|source=expression
type=('after'|'before'|'above'|'below'|'top'|'behind') desti-
nation=expression #SynsRelation
                                    ;
```

They can be used as:

```
TimeContainer A;
TimeContainer B;
ViewPort vp on A;
A before B;
A after D;
```

7.3.3 Validation

1- External validation

At the beginning, the interpreter creates a space container for each time container that is coming from the upper level. This helps to start from a point with less effort to create the GUI.

We wrote the following statement in the Prism language. These statements simply define a tabular placement for the search part. The statements are self-explanatory.

```
//rules on the elements inside fill container
syns:: fill.from_label before fill.fromSource;
syns:: fill.to_label before fill.toDest;
syns:: fill.date_label before fill.date;

syns:: fill.from_label above fill.to_label;
syns:: fill.to_label above fill.date_label;

syns:: fill.fromSource above fill.toDest;
syns:: fill.toDest above fill.date;

syns:: fill.complete before fill.rollback;
syns:: fill.rollback before fill.cancel;
syns:: fill.date_label above fill.complete;
syns:: fill.date_label above fill.rollback;
syns:: fill.date_label above fill.cancel;

syns:: displayElements above gridRoot;
syns:: pages0.Open before pages1.Open ;
syns:: pages0.Open above display.sortByPrice ;
```

The resulting GUI from these statements is depicted in figure 7.10. This figure depicts the GUI before executing these statements and after. The placement of elements is can be clearly noted.

2- Internal validation

The syntax-space model is concerned with four requirements in table 4.1. We explain how these requirements are satisfied.

1) Displacement elements are limited to enable navigation on the screen, inside the space container. Satisfied. They are created by viewports with a pre-defined behaviour.
2) Exclusively allow un-defining/enabling/disabling displacement elements. Satisfied. They are created by viewports with a pre-defined behaviour.

7.3.4 Summary

In this section, we introduced a linguistic syntax-time model, implemented and tested it.

We validated the model on a simple case study, and assessed internal validation by showing how it fulfils linguistic modelling requirements. We demonstrated how to produce a working GUI on the syntax-time level. We also demonstrated how the refinement of the GUI is carried out by refining the placement at this level.

The proposed linguistic syntax-space model is a proof of concept. In order to turn it into a real development tool, supporting tools should be implemented. For real life development, we need a designer tool to save the time in writing the Prism code.

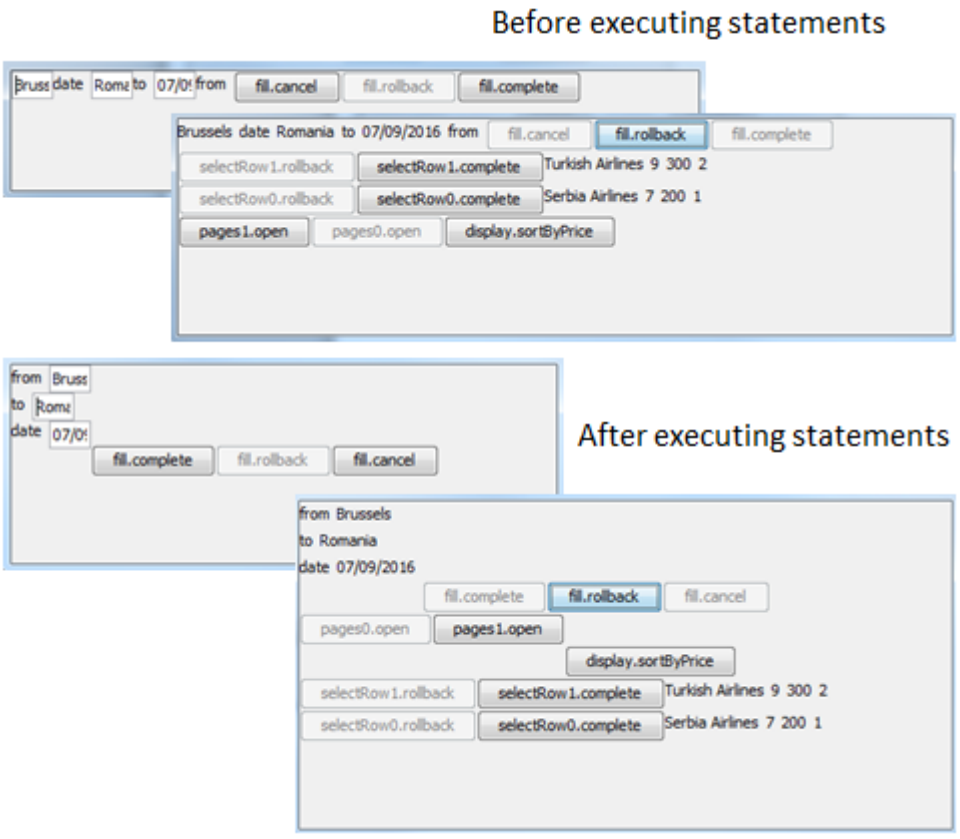

Figure 7.10 The GUI at the syntax-time level for the case study.

One may question the difference of the syntax-space model from layout manager systems, which are already implemented in existing development tools. The main difference relies in the approach. Existing layout managers build from bottom to top: they define placement rules on ex-

isting widgets, based on intangible knowledge in the mind of the developer/designer. Our approach is top to bottom: we place UI elements based on their meaning, which can be traced back to the task level.

For instance, in our approach, we can distribute UI elements on viewports (more precisely, on the viewport's underlying container). Viewports are defined semantically on how we want to divide the visible screen. We can say: this part is dedicated to display links (left navigation), while the other is to display a menu, and a third to display content. We can at upper levels attach the meaning of each UI element (using tags), and then enforce the distribution of UI elements based on this meaning. This guarantees that any concrete GUI will respect this distribution on the screen.

In the bottom to top approach, the meaning behind widgets and their distribution is in the developers mind. No guarantee if the developer changes, the same meanings are respected.

7.4 Modeling the widgets level

The purpose of the widgets model is to map UI elements from above levels to concrete widget.

7.4.1 The widgets model

1- Interfacing requirements

When concretizing widgets, we should not breach syntax-space rules. To guarantee this interfacing requirement, there are conditions on merging UI elements into a widget:

1) Merged UI elements belong to the same viewport.
2) When merging UI elements with a common parent space-container P, containers in the hierarchy between the widget and P cannot be concretized.

The second condition is important to guarantee the widget belongs to one and only one parent widget.

2- Defining widgets types

What widget type one can use to concretize UI elements? The widgets model requires a set of defined widgets types. To this purpose, we need

to provide the means to define a widget type.

The meta-model to describe a widget is depicted in figure 7.11. This meta-model allows defining new widgets types in the system. The figure contains an example on how we use this meta-model to define a widget type and how to use this widget type. We call a widget types as widget descriptor.

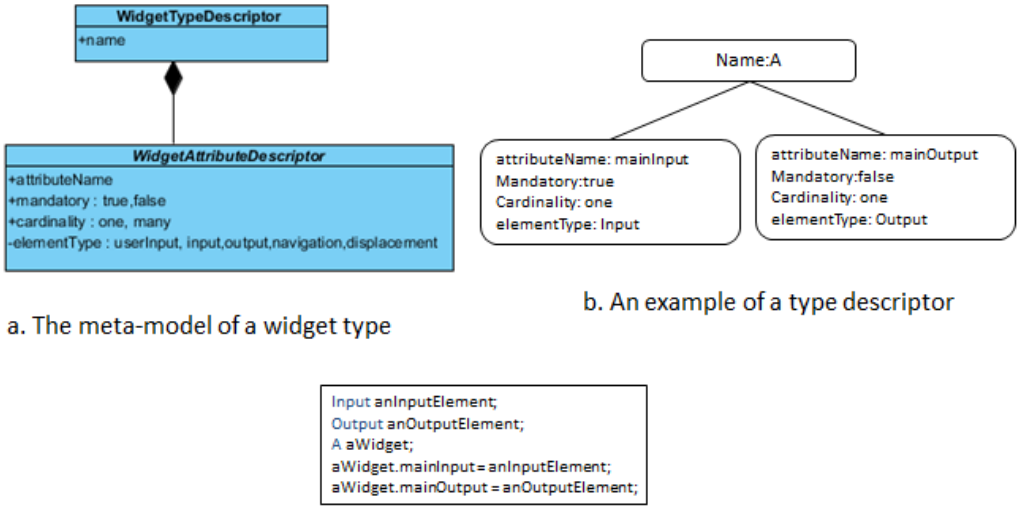

Figure 7.11 The meta-model to define a widget type

Notice in the example how we can define an attribute *"mainInput"* on the widget *"aWidget"* of type A. Defined attributes will be used in the mapping: A UI element can be mapped to one of the attributes of a widget.

3- Mapping rules of UI elements to widgets

General mapping rules to respect when mapping are:

1) A widget is a visible object on the screen.
2) A widget can map to several UI elements, with respect to its descriptor. The widget is responsible on rendering the UI element as a visible object or not.
3) User input elements (for filling data from the user) cannot be mapped to a widget as invisible. Therefore, a user input element can map to an attribute of type userInput only. This is to ensure that the widget is well informed on user inputs, and it is up to it

to make it visible.

At this level, we may prefer to use the same widget to concretize different visible UI elements together. If the widget has only one attribute for output elements, it will not be able to map different elements together. Therefore, we define a merge operation among UI elements to produce one UI element.

Anyway, there are conditions on merging UI elements at this level:

1) Output UI elements can be merged into one output element at this level. The resulting UI element can concatenate values of these outputs together.
2) Input elements (non-user input elements), navigation elements and displacement elements can be merged together if they do not conflict with each other. This can be validated at run-time as: only one element is active at a time.

In practice, there are many different widgets in programming languages. To use a concrete widget on our widgets model, we need first to create a widget descriptor for it. Once the widget descriptor is in the system, we can map UI elements to its properties. We will elaborate more on this issue in the next section.

4- Mapping to the real GUI system

Once UI elements are mapped to widgets, the model is ready to be rendered on a graphical system. Until now, our models on all levels are independent from the platform.

Each widget descriptor defines attributes on a widget to map UI elements to. These attributes should be mapped to real graphical components on the GUI system. This is a one-to-one transformation from linguistic widgets to concrete widgets in the target platform.

However, a mapper is needed to do this transformation. It will map events on the graphical system to input elements in the linguistic model. Each platform should implement its own mapper. This mapper should be accessible to the widgets level to make use of it.

7.4.2 Implementation

In order to reduce the amount of work in this proof of concept, we did not support defining widget types (widgets descriptors) in the Prism lan-

guage. We depend on Java classes to define these descriptors and inject them in the interpreter. Anyway, implementing this feature in the Prism language will add capability more than value to the proof of concept.

Two statements are implemented in prism. The first statement concretizes and maps UI elements to a widget of a type. The other is the selector statement we have already seen before.

```
lexStatement:
  'concretize' uiElement=qualifiedName'as' widget-
Type=Identifier '<' concretizePropertyList '>'
  |'select' e=Identifier 'from' list=expression 'where' con-
dition= expression block=lexBlock;
```

An example on using the concretize statement is:

Concretize fill.complete as Button <>;

7.4.3 Validation

1- External validation

The interpreter provides a default mapping of UI elements to widgets. Every output element is mapped to a label, every user input element is mapped to a text box, while other input elements are mapped to buttons. Space containers can be mapped to panels, at least one is needed for the root. The interpreter can produce a GUI like the one on the syntax-space level, depicted in figure 7.10.

When the search button is clicked, text boxes are replaced by labels. This is the default behaviour of the interpreter. We want to change this by telling the interpreter to use a text box even when the element is disabled.

Second, the rollback and the cancel are alternating. One is appearing at a time. Therefore, we created a custom widget called "XORButton" that concretizes both.

The statements to do the above are depicted below. Notice the changes on the GUI that are depicted in figure 7.12.

```
lex:: concretize fill.fromSource as TextBox.onActionPerformed <>;
lex:: concretize fill.toDest as TextBox.onActionPerformed <>;
lex:: concretize fill.date as TextBox.onActionPerformed <>;
lex:: concretize fill.rollBack as XORButton.onActionPerformed <alternate=fill.Cancel>;
```

Figure 7.12 The GUI at the lexical (widgets) level for the case study.

2- Internal validation

We elicited linguistic modelling requirements in table 4.1. The linguistic widgets model is concerned with four of them. All these requirements are satisfied fully or partially. We explain how these requirements are satisfied. Note that the satisfaction of the requirement on all levels is already discussed on the task level, so we will not get back to it.

1) **Do not allow adding an input widget**: Partially satisfied. There is no way to fully satisfy it because the model at the widgets level allows mapping not defining. Concrete widgets may define new widgets internally.
2) **Allow mapping UI elements from the upper level with widgets**: Satisfied using the **"concretize"** statement.
3) **Shall have a meta-description of possible inputs and outputs it can accept**: Satisfied using widget type descriptors.
4) **UI elements relation with widgets is of type many-to-one**: Satisfied with respect to the widget type descriptor.

7.4.4 Summary

We have been pragmatic in modelling at this level by supporting concrete GUI widgets through widget type descriptors. Although powerful,

it carries out challenges to the complete linguistic perspective.

Components are programs. When we use these components in our linguistic models, we are introducing hidden parts to the program. They can breach all the constraints we elicited in table 4.1 on modelling requirements. For instance, a custom widget can introduce different input elements, while input elements are introduced at the right level of abstraction.

To avoid such a challenge, we should go for linguistic widgets. These are widgets that are built from a linguistic perspective and a linguistic model. This is a main concern before starting to use the GUI linguistic model in products.

7.5 Modeling the widgets properties level

This is the last model that will style the GUI. Now everything is done and we still need to add final touches to it.

7.5.1 The widgets properties model

1- Interfacing requirements

The only important interfacing requirement is not to affect any aspect defined on the upper level. This imposes restrictions on the set of attributes to define.

7.5.2 Implementation

We define a set of attributes on all widgets. The set of attributes here is by no means complete, and should be enlarged to allow adding more styling capabilities. The set of attributes defined is depicted in table 7.1.

Implementation is straightforward. We need a statement to select an element (or a list of elements) and another to assign a value to an attribute. We have already presented the "select" statement before. Now we use it to select elements. The formal grammar rules for statements on the widgets properties are:

```
alphabeticalStatement: assignStatement
 |'select' e=Identifier 'from' list=expression 'where' condi-
tion=expression block=block;
```

An example on using the above statements is given below. Note that to

manipulate widgets properties, we use a dedicated list: "uiWidgetsList".

```
alph::select e from uiWidgetsList where true { e.ForegroundColor="red";};
```

Font	**Value**: the value of the UI element	**Height**
FontStyle: bold, italic, underline…	**MarginX**	**Align**: left|right|center
ForegroundColor	**MarginY**	**VerticalAlign**: top|bottom|center
BackgroundColor	**Width**	**Borders**: thikness of borders

Table 7.1 Attributes of widgets in the widgets properties model.

7.5.3 Validation

2- External validation

We made different styling the GUI at the previous levels to get the result depicted in figure 7.13. The figure shows a completely customized GUI for the search screen, while the second screen with search results needs more refinement.

Prism statements used to do this refinement are also depicted in the figure. They are self-explanatory.

3- Internal validation

We elicited linguistic modelling requirements in table 4.1. The linguistic widgets properties model is concerned with four of them. All these requirements are satisfied based on the selected list of attributes in table 7.1. We explain how these requirements are satisfied. Note that the satisfaction of the requirement on all levels is already discussed on the task level, so we will not get back to it.

1) **Shall not define a property to hide a widget**: Satisfied. See table 7.1.
2) **Shall not define a property to enable/disable a widget**: Satisfied. See table 7.1.
3) **The coordinates of a widget are relative to the containing container**: Satisfied. The semantic of setting x,y on this level is

interpreted as relative to the container.

4) **The coordinates of a widget are validated against placement rules at run-time**: Satisfied. See the syntax-space level rendering algorithm.

```
//set the foreground color of all elements to blue.
alph:: select e from uiWidgetsList where true { e.ForegroundColor="blue";};
//set the label, forecolor and position of the fill.complete buttton.
alph:: select e from uiWidgetsList
  where e.id=="fill.complete"
    { e.Width=250;e.value="Search";e.X=100; e.Y=75; };
//set width of text boxes.
alph:: select e from uiWidgetsList
  where e.id=="fill.from_label" || e.id=="fill.to_label" || e.id=="fill.date_label"
    { e.Width=100;};
//set width of labels.
alph:: select e from uiWidgetsList
  where e.id=="fill.fromSource" || e.id=="fill.toDest" || e.id=="fill.date"
    { e.ForegroundColor="blue"; e.Width=200;};
//set label, width, position and forecolor of he cancel button.
alph:: select e from uiWidgetsList
  where e.id=="fill.rollback"
    { e.ForegroundColor="red"; e.Width=100; e.value="Cancel"; e.Y=75; e.X=400;};
```

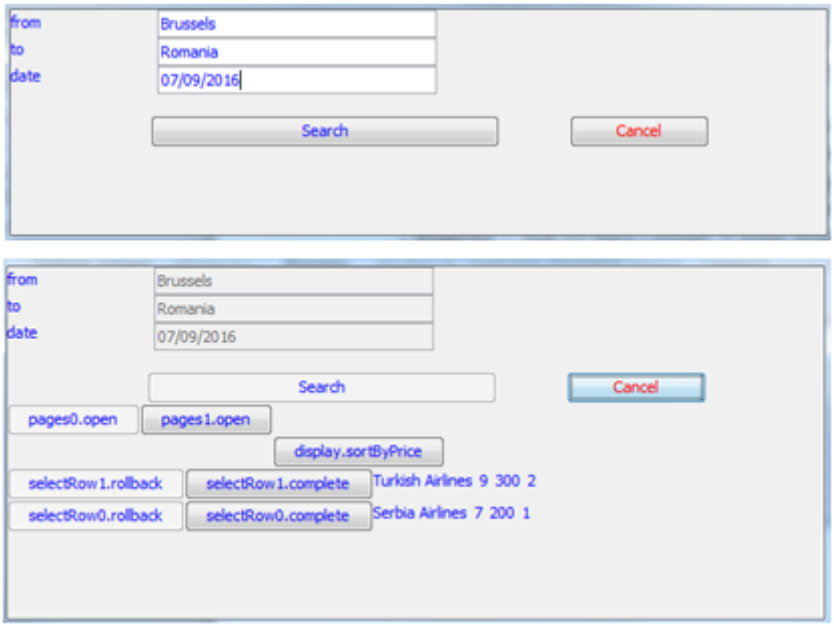

Figure 7.13 The Final GUI at the alphabetical level.

7.6 Chapter Summary

In chapters 5, 6 and 7 we have presented a model on each linguistic level. The group of all these models forms the linguistic model to develop the GUI.

The models presented and demonstrated step-by-step on a simple case study. The case study provides a proof of concept that a complete GUI can be built using modelling approaches from the linguistic perspective.

Chapter 7. A GUI Linguistic Model: Perceptual Levels

Chapter 8 Assessing the GUI Linguistic Modeling Approach

8.1 Context

In chapter 2, we elicited shortcomings in different approaches. In chapter 3, we **solved** the root cause of these limitations. In chapters 5, 6 and 7, we introduced our linguistic models. In this chapter, we assess the impact of solving the root cause on these approaches, in the light of the linguistic modelling approach.

Concretely, we answer the following questions[36]:

1) How the shortcomings in the software development tools approach are addressed by the linguistic modeling approach?
2) How the shortcomings in UI modeling approaches are addressed by the linguistic modeling approach?
3) How the shortcomings in the task models approach are addressed by the linguistic modeling approach?

A section is dedicated for each approach. We re-visit shortcomings identified per approach. For each shortcoming, we provide a score of the perception we have about the elimination of the limitation. The score has three values:

1) Solved: the limitation does not exist anymore in the linguistic modelling approach.
2) Highly addressed: there are still issues about these limitations.

[36] Methodologically, we should state questions in the reverse order. The reason we present them in this order is to enhance readability for the reader, because what is learnt on a question will help to better understand the discussion on the next one.

3) Addressed: main obstacles are removed towards solving the limitation.

We conclude the chapter with an updates progress map.

8.2 UI development tools approach

The Prism programming language is a UI development tool. We have already proved the ability of this language to produce a working GUI. Let's assess shortcomings in the UI development tools approach using the Prism development language.

8.2.1 Twisting UI code with other modules

Using the Prism language, the UI code is separated from other modules. The UI code is integrated with other modules through detailed functions, which are at the semantic level. Note that:

1) The semantic model is isolated from other models and cannot manipulate them.
2) The semantic level's role is to create UI elements and map them to the data model.

Therefore, other modules cannot modify any UI aspect intentionally, with the exception of adding/removing UI elements per-detailed function.

We can state that the Prism programming language provides a **higher degree of** separation between the UI code and other modules code (other software engineering modules), in comparison with all existing UI development tools

The main reason behind this separation in code is the univocal separation of concepts per level (Req_1.3.1) that is satisfied and implemented in the Prism programming language.

Remaining issues in this limitation are related to the semantic model. A good model should provide complete separation between UI code and other modules. Our proof-of-concept semantic model uses Java, which provides more capabilities than needed at this level. A semantic model that is limited to what is allowed at this level can eliminate this imitation completely.

Score: Highly addressed.

8.2.2 No separation of concern

The Prism programming language provides **a high level of** separation of concerns than existing UI programming languages.

In prism, input elements are not defined on the same level. Adding a new input element can work on the appropriate level. Therefore, a grid component is not on the same level of abstraction as a button. It spreads over several linguistic levels.

A button cannot replace a grid because the button does not have the same type of descriptor as the grid. The grid (with pagination) type descriptor declares that it should be mapped to navigation elements to enable navigation. A button descriptor does not have such a constraint.

The decision to accept replacing a widget with another is based on comparing their type descriptors (equivalences). A type descriptor defines the concerns in the widget. Therefore, separation of concerns is assessed.

Notice that separation of concerns can be enhanced better if linguistic widgets are used instead of custom components provided by programming languages. Therefore, to completely eliminate this limitation, we need to use linguistic widgets to replace GUI components.

Again, the main reason behind this separation in concerns is the univocal separation of concepts per level (Req_1.3.1).

Score: Highly addressed.

8.2.3 Modularity may lead to repetition

The Prism programming languages **allows no repetition**. The DRY principle is satisfied.

A UI concept belongs to one and only one level. Activities are also separated between levels. Therefore, there is one place the carry out an activity.

Score: Solved.

8.2.4 Low traceability with requirements

The Prism programming language provides the **highest level of** traceability of any known in any UI development tool. Every element at any level can be traced back to the task level.

The main reason behind this high traceability is enabling the UI development since the analysis phase (Req_1.1), which is satisfied and implemented in the Prism programming language.

Score: Solved.

8.2.5 Assessment results

The Prism programming language solves to a high degree shortcomings in existing UI development tools. This is basically due to the satisfaction of two requirements:

1) A concept belongs to one and only one level (Req_1.3.1).
2) The UI development is enabled since the analysis phase (Req_1.1).

Table 8.1 illustrates limitations and assigned scores.

Limitation	**Score**
Twisting UI code with other modules	Highly addressed
No separation of concern	Highly addressed
Modularity may lead to repetition	Solved
Low traceability with requirements	Solved

Table 8.1 Personal Subjective elimination scores of limitations in the UI development tools approach.

8.3 Modeling Approaches

Identified limitations in this approach are hierarchical. We assess them from the deeper cause until we reach superficial shortcomings.

8.3.1 The technical UI is methodologically weak

The technical domain in the linguistic model, as depicted in the prism programming language, is methodologically strong. It can start from the analysis phase and does not need any other models to fulfil the analysis gap. This limitation is **solved**.

Score: Solved.

8.3.2 The coupling problem and the UI dilemma

This problem is **solved** by adopting option 2 in figure 2.16.

We explained in section 2 (page 42) that implementing option 2 solved the UI dilemma by negating the fact TECH_MOD_2 (Technical UI models are methodologically weak). Anyway, we also mentioned that it would change the role of HCI models. So what is the role of HCI model in the linguistic UI development?

The new role of HCI models in the UI development is defined per level. Recall that HCI models are a formalism of usability knowledge. In chapter 4, section 4.4.2, we discussed a potential classification tool for usability guidelines. The same can apply on HCI knowledge formalized in HCI models. More concretely, the translation of HCI concepts is performed per level as follows:

1) Linguistically classify each HCI concept using the Linguistic Prism (figure 4.8). This would designate the area of impact of these concepts.
2) Translate the HCI concept into UI concepts in these levels.

The new role of HCI models is supporting and no more a key player.

Finally, in chapter 4, section 4.4.3, we explained how the UI evolves on different linguistic levels. We also explained how the UI could evolve based on adaptations to the context of use (figure 4.9). HCI models can model the context of use and therefore lead this evolution process, by determining what fragments of code to execute per-level.

Score: Solved.

8.3.3 Complex transformation engines

The concept of transformation engines becomes a bit ambiguous in the

linguistic model.

Transformation engines in general transform a source model to a destination model. In the special case of UI models, the ultimate goal of using transformation engines is to generate the final UI.

The linguistic model produces the final UI at the widgets and widgets properties level. It substitutes transformations with realizers: refinement from each level to the lower. Realizers are not transformation engines, because the lower level cannot be generated from the upper one.

The real transformation in the linguistic model happens at the widgets level, where we transform widgets on the widgets level into concrete widgets in the graphical system. This transformation is **highly simplified** because it is a mapping between a widget concepts and a concrete graphical component. This mapping is defined using widget type descriptors.

Besides, a transformation engine from a UI model to a linguistic model would probably take the form of reverse engineering. This is due to the potential conceptual gap between these models and the linguistic one. The first stone in the linguistic model is to define tasks. In reverse engineering, that would be the first step.

A transformation from one linguistic model to another looks of low interest. Recall that the linguistic modelling framework accepts different models and notations at the same level (figure 4.10). Therefore**, two linguistic models can co-exist without the need to transform one to another**.

A transformation from an HCI model to the linguistic model is much enhanced. This is because the transformation **is sliced into six transformations**: one at a level. Recall that the GUI model is sliced on levels, where UI concepts belong to one and only one level. Therefore, each slice of the transformation engine is concerned with a limited set of concepts that belong to that level. Figure 8.1 depicts the slicing the transformation engine.

Score: Highly addressed.

8.3.4 Limited capabilities

The overall capability of the linguistic model is the sum of capabilities on

each level. We can discuss the capability of the task model, semantic model, the navigation model, the placement model, the mapping with widgets model and the widgets properties model. The capabilities of each model can be addressed in isolation of other models. Therefore, the linguistic model has a **great potential to enhance its capabilities**.

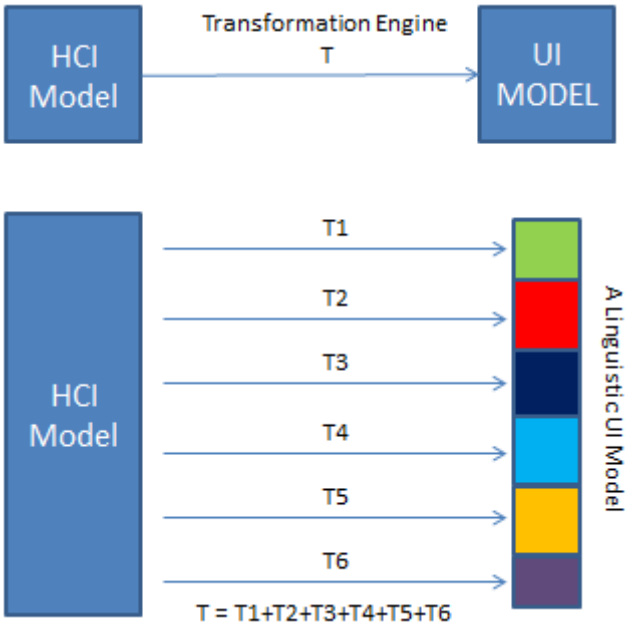

Figure 8.1 The slicing of transformation engines from HCI models into linguistic GUI ones.

Enhancing the capabilities of a model on a level may increase complexity. In linguistic models, this is not a big concern because we already have simple models on levels (thanks to the separation of concerns). Recall also that the linguistic modelling framework allows the co-existence of different models per-level. This means we can use two notations: a simple notation for simple cases and another more advanced for special cases.

Score: Highly addressed.

8.3.5 Limited types of generated UIs

Although we declared since the beginning that the linguistic model ad-

dresses the GUI, but it can be used also to generate other types of UIs. For instance, if we use vocal widgets instead of graphical ones, we can produce a vocal UI. Eventually, the sequential nature of vocal UIs can be considered in the navigation model (syntax-time). Anyway, as we did not provide a case study on this type of UIs, we consider this limitation as **addressed**.

Score: Addressed.

8.3.6 Assessment results

Existing limitations in modelling approaches do not look that limiting in the linguistic GUI model. This is mainly due to having several levels of abstraction and isolation of concepts per-level. Besides, root causes of these limitations are solved. Therefore, it is expected to decrease the impact of these limitations.

Table 8.2 illustrates limitations and assigned scores.

Limitation	Score
The technical UI is methodologically weak	Solved
The coupling problem and the UI dilemma	Solved
Complex transformation engines	Highly addressed
Limited capabilities	Highly addressed
Limited types of generated UIs	Addressed

Table 8.2 Subjective elimination scores of limitations in UI modeling approaches.

8.4 The task models approach

The linguistic task model is compared to UI task models.

8.4.1 UI task models have a methodological problem

The methodological problem in UI task models is **solved** in the linguistic task model by limiting the role of the task model on the analysis phase. The toll is paid: find a task decomposition stopping criteria that limits the role of the task model on the analysis phase. The criteria employed are those defined by Guerrero [Guerrero2008].

The methodological problem is solved due to (1) enabling the UI development since the analysis phase (Req_1.1) and (2) using concepts that belong to the technical UI domain only (Req_1.2).

Score: Solved.

8.4.2 UI task models are not adopted in real UI development

This limitation is **addressed**. We provided a working example in the case study on search for a flight. The difference between the example in our case study and other examples on UI task models is that the task model is employed at the analysis phase without limiting he design space on lower levels.

Anyway, more examples should be conducted using the linguistic model to proof its applicability in real development projects.

Score: Addressed.

8.4.3 Assessment results

The linguistic task model does not have a methodological problem like in UI task models. It is promising to be employed in real development projects.

Table 8.3 illustrates limitations and assigned scores.

Limitation	Score
UI task models have a methodological problem	Solved
UI task models are not adopted in real UI development	Addressed

Table 8.3 Subjective elimination scores of limitations in UI Task models approaches.

8.5 Chapter Summary

In this chapter, we assessed the impact of different limitations on the linguistic GUI model. We re-visited shortcomings in each of the approaches identified at the beginning of this chapter and discussed how the limitation is addressed/solved in the linguistic GUI model.

Chapter 8. Assessing the GUI Linguistic Modeling Approach

This chapter closes the second milestone. As a result, 5 limitations are solved, 3 are highly addressed and 2 are addressed. Figure 8.2 depicts the progress map at the end of the second milestone. The elimination score of each limitation is depicted in the degree of the blue colour. The darker the blue colour is, the higher the score of elimination is.

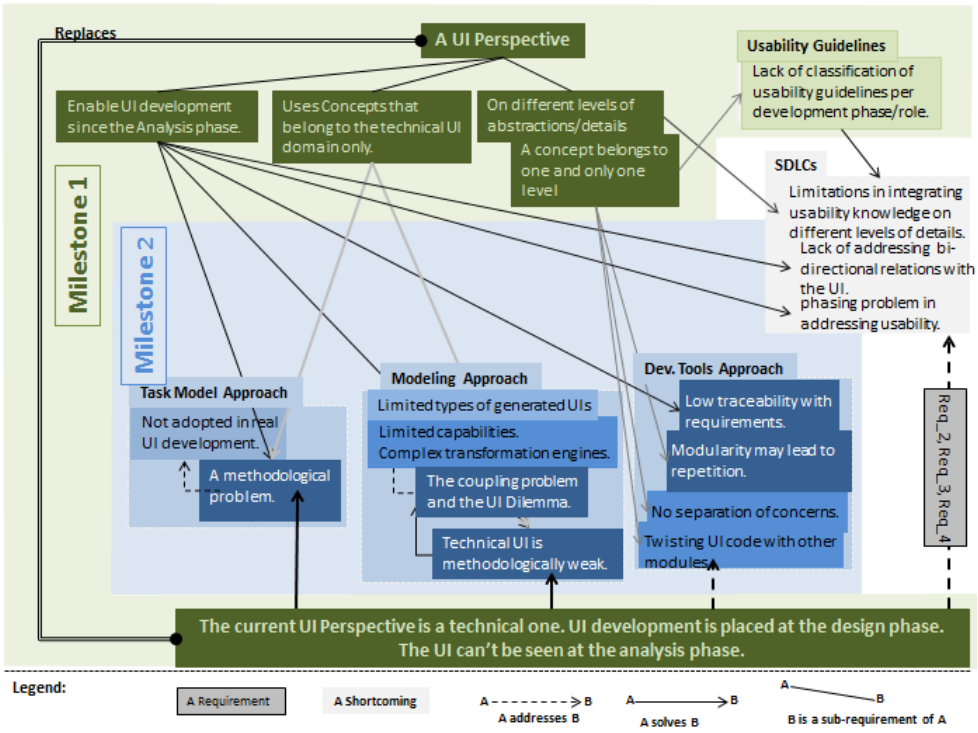

Figure 8.2 The progress map at the end of the second milestone.

Chapter 9 The Prism Development Life Cycle

9.1 Context

The linguistic model enables a structured development of a concrete GUI since the analysis phase. This is demonstrated until chapter 8. The structured development of the concrete UI induces a UI development life cycle with its phases aligned to software development ones.

This chapter is dedicated to present the UI development life cycle that is induced from the linguistic structured GUI development. We call this UI development life cycle as the Prism-DLC. It also explains how to enact this DLC and assesses how requirements Req_2, Req_3 and Req_4 can be fulfilled.

The prism UIDLC is different from common. It is based on a linguistic perspective to the development of the GUI. It mainly addresses the integration between HCI and Software Engineering in the development of a software product with usable UIs, with focus on the evolution of the software, while maintaining high traceability from high-level concepts to concrete refinements.

9.2 The Prism-DLC

Prism is a UI development life cycle that aligns the linguistic UI development and the software development. The main difference from other DLCs activities is the use of a classification step to analyze and classify UI requirements in order to determine the level(s) of enactment.

The linguistic perspective allows perceiving the UI since the analysis phase. This allows defining the term "**UI requirements**" based on the principle: "*any modification(s) on the UI is based on a UI requirement*". This principle is of high importance for traceability reasons. It is more important to trace the modification (where it comes from) than to link to the rationale (why do they come)

UI requirements may come from usability (like adapt to the user's culture), software design decisions (software modules and interaction capabilities), and software detailed design modules (like allow the user to interact with a function in the system).

The impact of a UI requirement is classified using the Linguistic Prism classification tool to decompose it on linguistic levels, which is grouped in a UI patch. A **UI patch** is *the impact of a UI requirement on different linguistic levels*.

Prism does not impose any constraint on the software DLC to integrate with. Anyway, in order to explain how integration and development is performed in Prism, we show integration with general development phases: analysis, design, detailed design, coding and testing. It is up to the SDLC implementer to adapt Prism to the specific needs of the enacted lifecycle.

In the following, we explain the Prism-DLC on each phase and finally present the complete lifecycle.

9.2.1 Prism at the Analysis phase

The system development starts with the analysis activity of the system. When the analysis activity is completed (as defined per the applied SDLC), requirements pass through the classification phase to create the UI patch. This UI patch allows creating the task model from the software analysis.

The UI patch, created after the software analysis, may impact several linguistic levels. Note that **software requirements are expressed at different levels of details**. A user may express very detailed requirements like the preference for a specific theme of colors. The classification activity identifies UI-related aspects in every requirement and maps them to the appropriate linguistic level.

Notice that the classification happens on all requirements. The Linguistic Prism disperses requirements that are not related to the UI.

While developing the task level, usability shortcoming in the analysis might be identified. The feedback loop from the task level to the analysis phase, not only ensures that usability requirements are gathered, it also assesses consistency between the task model and the system analysis. It reflects the impact of the UI on the analysis phase.

After the analysis is completed, the UI is fixed. This version of the UI might be communicated with the user as a premature version of what is expressed in requirements. Later modifications on the UI should not affect this version, which we call: the **analysis-UI** version. Modifications to this version should be communicated/approved with the user first.

Having a UI at the analysis phase enables testing usability regarding the task model. What we need to be sure of at this phase is that the task model is correct and the interaction at the task level corresponds to the end-user's expectations.

With suitable linguistic tools, it is possible to develop a "linguistic prototype" that reflects user's requirements on lower levels. This might help the user to tell after s/he sees it (the IKIWISI syndrome). The difference from common prototypes is that the linguistic prototype becomes part of the fixed UI, not a throw away.

The Prism-DLC at the analysis phase is depicted in figure 9.1.

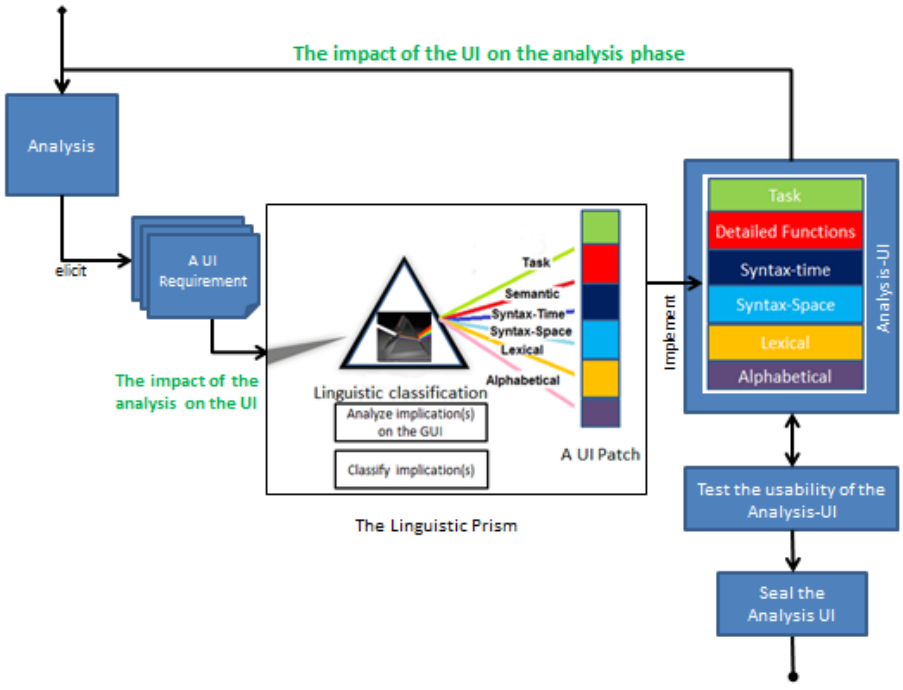

Figure 9.1 The Prism-DLC at the analysis phase

9.2.2 Prism at the design phase

With the linguistic task fixed after the analysis phases, the design phase starts with immediate feedback from the UI. The analysis UI may have gathered usability requirements on the design of the system itself. The design should consider the feedback from the analysis UI since the beginning.

The design phase makes design decision on the domain model, which affects the identification of UI elements in each task. Anyway, system design decisions become UI requirements because they carry out modification on the UI. These UI requirements are again linguistically classified to identify their impact on linguistic levels. A UI patch is also created and implemented.

The feedback loop from the semantic level to the design level is present to ensure that the UI and the design are consistent. It also reflects the impact of the UI on the design phase

If the UI patch at the design phase contains implications on the task level, this depicts a shortcoming in requirements: Tasks were not identified properly. If the software life cycle can handle such incompleteness, an alert can be triggered and managed to solve this shortcoming appropriately.

At the detailed design phase, the same repeat as with the design phase. After completing the detailed design, the semantic level is fixed. The UI for carrying out tasks has a complete set of UI elements. No further modifications are allowed on the semantic level without repeating the design and detailed design phases. This version of the UI is called the **design-UI**.

Having a UI at the design phase enables testing usability regarding the semantic model. What we need to be sure of at this phase is that the semantic model is correct and the interaction at the semantic level corresponds to the end-user's expectations.

With suitable linguistic tools, in addition to system design tools, it is possible to develop a "linguistic prototype" that reflects user's requirements on lower levels. The Prism-DLC at the design phase is depicted in figure 9.2.

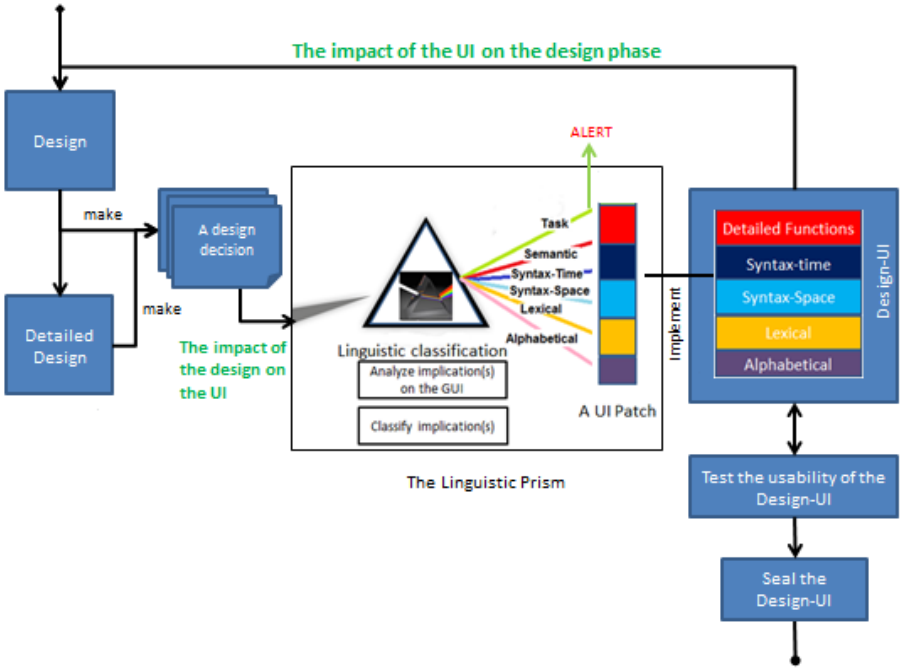

Figure 9.2 The Prism-DLC at the design phase

9.2.3 Prism at the implementation phase

In parallel with the implementation phase, the UI can be refined on the syntax-time, syntax-space, widgets selection and stylistics on the widgets properties level. This gives the UI design the freedom to manipulate these aspects with the guarantee that any implemented design is compatible with the semantic and the task levels. Both activities are synchronized to start the testing activity.

The software implementation phase has nothing to provide to the UI. Everything the UI needs is agreed on the semantic level. The software implementation phase and the Prism activities (from the syntax-time level to the alphabetical level) need to synchronize at the testing phase to ensure that the implementation meets the design.

The Prism-DLC at the implementation phase is depicted in figure 9.3.

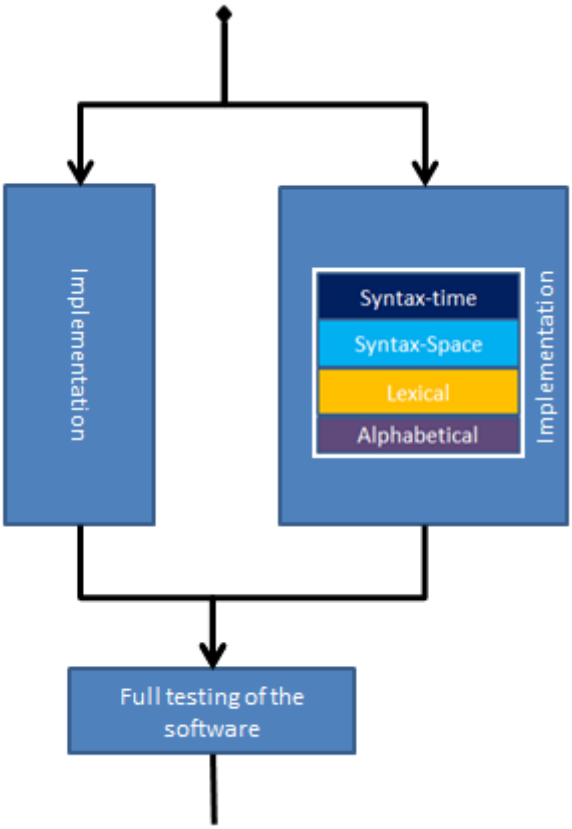

Figure 9.3 The Prism-DLC at the design phase

9.2.4 Prism at the testing phase

Testing can be decomposed into two activities: Validation is to assess the implementation conforms to the specification (the design), and Verification to verify that the product satisfies user's requirements. Note that validation testing can be done on the design-UI version and verification can start on the analysis-UI version. UI Validation testing is to compare the design-UI version with the final UI version on the navigation design, placement, widgets and stylistics. Functionality is guaranteed.

With testing already started on previous phases, what is left is a full testing of the system to ensure mainly validation of the system code. A full usability testing can be performed on stable versions to ensure fewer disturbances to the user. Usability at this level is concerned with minor is-

sues on the UI that do not affect the design or the analysis, which are fixed before. Even though new requirements might be elicited during the test with the user, it is up to the enacted development life cycle to handle such situations.

Finally, the general Prism-DLC is depicted in figure 9.4.

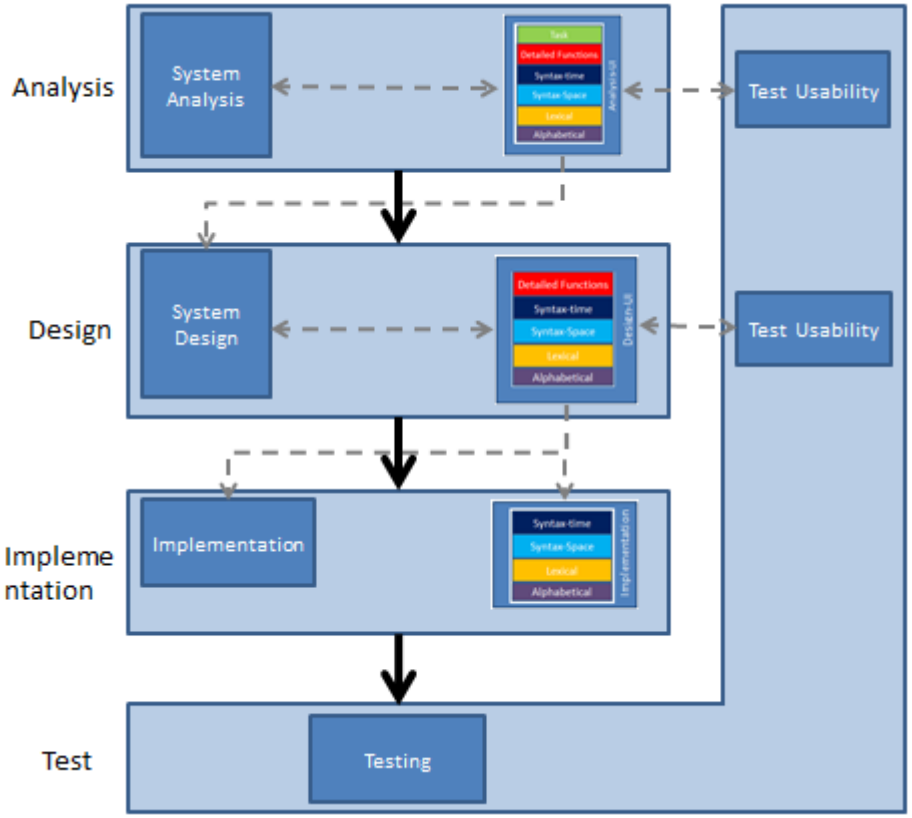

Figure 9.4 An overview of the Prism-DLC

9.3 The Prism enactment method

1) During the analysis phase
    a. Analyze the system.
    b. Linguistically classify each requirement and create a UI patch.
    c. Implement the Analysis-UI.

   d. Test usability with the end-user (or representing role).
   e. Check consistency between the Analysis-UI and the system analysis.
   f. Repeat until consistency is verified.
   g. Seal the Analysis-UI.
2) During the design phase.
   a. Design the system.
   b. Linguistically classify every design decision. Classification on the task model is not allowed.
   c. Implement the Design-UI.
   d. Test usability with the end-user (or representing role).
   e. Check consistency between the Design-UI and the system design.
   f. Repeat until consistency is verified.
   g. Seal the Design-UI.
3) During the implementation phase. Activities can be executed in parallel:
   a. Implement the system
   b. Update the UI on the syntax-time, syntax-space, widgets and widgets properties.
4) During the testing phase:
   a. Test the complete system.
   b. Full test of the usability of the UI.

9.4 Assessment

It is very hard to validate a development life cycle. This requires huge resources that are not available in this thesis. Besides, testing the Prism-DLC requires also linguistic UI tools (graphic designers for every linguistic level) that are not available to the time. The Prism programming language is not ready for real development yet. It served as a proof of concepts of a linguistically structured UI development.

Anyway, in order to increase the motivation to support further research and validation of the Prism-DLC, we assess our elicited requirements on the development lifecycle to align UI development and software devel-

opment activities.

9.4.1 Enhance the integration of usability knowledge on different levels of details

The Prism-DLC allows integration of usability knowledge on different levels of details. Even at the analysis phase, when the user expresses very detailed and low-level requirements, it is possible to consider in the Analysis-UI.

Therefore, Prism-DLC **is promising to fully satisfy** requirement Req_2.

9.4.2 Addresses the bi-directional relations with the UI

This is one of the most important properties in the Prism-DLC. It decomposes the bi-directional relation into two bi-directional relations, each on a level. The relation of the analysis is bi-directional. The relation with the design is also bi-directional. But the difference from other SDLCs relies in the absence of relations from lower levels to upper ones.

Figure 9.5 depicts the change on the nature of the bi-directional relation between the analysis and design from one part and the UI development from the other part. Therefore, the Prism-DLC is **promising to solve the bi-directional relation with the UI**.

9.4.3 Address the phasing problem when addressing usability

The Prism-DLC enables assessing usability since the Analysis-UI. Usability testing can also be carried out on the Design-UI. In fact, usability can be assessed at any level, as the linguistic GUI model allows producing a GUI at any level.

Therefore, integrating the Prism-DLC in agile methods **is promising to avoid the phasing problem**. Usability can be assessed during the iteration and at the end of it.

9.5 Chapter Summary

This chapter presented Prism-DLC, the induced development lifecycle from the structured linguistic development of the GUI. It shows how such a DLC can solve shortcomings in existing SDLCs concerning usa-

bility integration in the development phases of an information system. The Prism-DLC is very promising to fulfill the three requirements we identified in chapter 2.

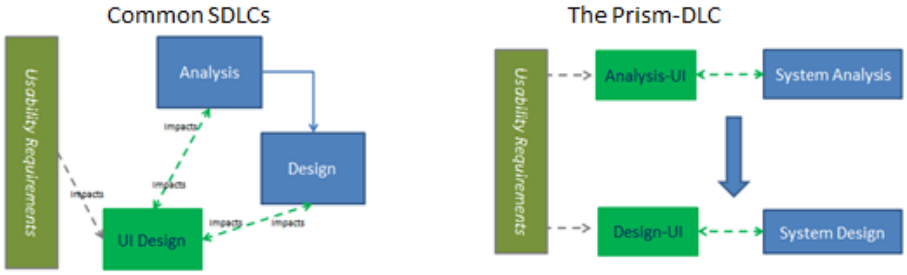

Figure 9.5 The Prism-DLC solves the bi-directional relation with the UI.

With this chapter, we reach to the last milestone in our solution process. The final progress map at the end of the milestone 3 is depicted in figure 9.6.

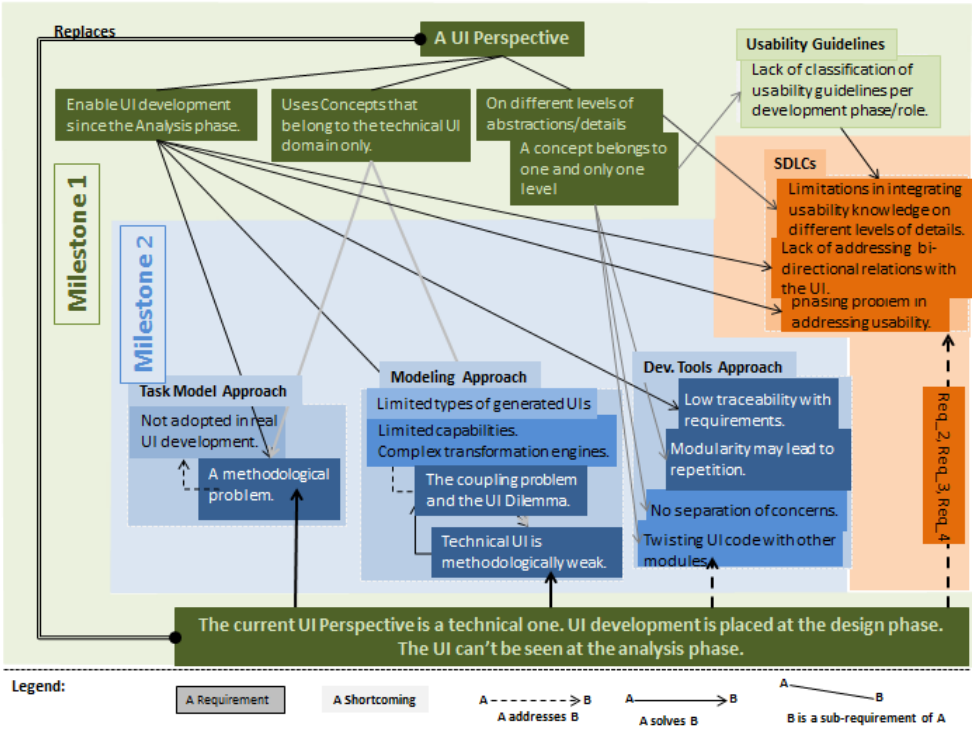

Figure 9.6 The Prism-DLC solves the bi-directional relation with the UI.

Chapter 10 Towards a Linguistic GUI Paradigm

This chapter is dedicated to argue that based on the contributions of this thesis, which we have presented in the previous chapters; we are standing a step away from a new paradigm for the GUI development.

We start with a discussion on paradigms using the Kuhn Cycle [Kuhn1996] for the evolution of science, position our contributions on this cycle and finally we start presenting how different concepts in HCI can be addressed in the new paradigm.

10.1 The Kuhn Cycle

New fields start in the "pre-Science" step. Fields start to focus on a specific area but are unable to solve it.

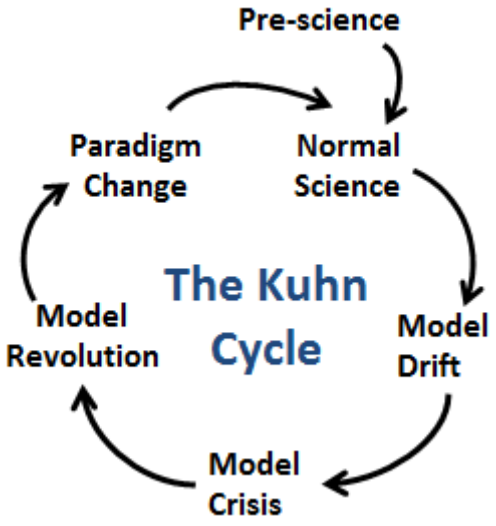

Figure 10.1 The Kuhn Cycle

When efforts become fruitful by providing a model of understanding of

the problem and provide solutions to major problems, the "Normal Science" step begins.

Over the time, this model of understanding becomes limited as the field digs deeper into its area of interest. New questions arise and the model becomes weak to provide the right answers. This is the "Model Drift" step.

When unsolved anomalies increase more and more, the "Model Crisis" step is reached. It is a crisis because decisions are not made rationally. Guess works and intuition are used instead.

A "Model Revolution" step happens when a new model of understanding is formed. It is a revolution because the new model is the new paradigm that is radically different from the old one. **Believers in each paradigm cannot communicate well**.

Once the new paradigm is settled, the "Paradigm Change" step begins. The field's transition from the old to the new paradigm happens while maturing the new paradigm. When the old paradigm is sufficiently replaced, the normal science step starts. The cycle starts all over again.

Kuhn defined paradigms as *"universally recognized scientific achievements that, for a time, provide model problems and solutions for a community of researchers,"* [Kuhn1996, page X]. A paradigm describes:

1) What is to be observed and scrutinized.
2) The kind of questions that are supposed to be asked and probed for answers in relation to this subject.
3) How these questions are to be structured.
4) How the results of scientific investigations should be interpreted.

A shorter version of this definition would look like: *"a paradigm is a comprehensive model of understanding that provides field's members with viewpoints (perspectives) and rules on how to look at the field's problems and how to solve them."*

10.2 Decoding the HCI and SE Evolution

Since chapter 1, we focused on the human factors crisis because it is the problem that Software Engineering (SE) in the Computer Science cycle could not solve. Based on the visit to the history in chapter 2, the emergence of HCI can be decoded using Kuhn Cycle as follows:

1) In the late 70s, SE found itself in the middle of the *"Model Crisis"* step on the cycle of Kuhn because it was unable to solve the human factors crisis.
2) Some scientific fields, that constituted later what is known now as Cognitive Science, provided a new model of understanding to address the human factors crisis. This depicted a *"Model Revolution"* step, where different fields of research coalesced to form the HCI discipline. Computer Science was one these fields.
3) The newly introduced model of understanding got the attention of many researchers. The *"Paradigm Change"* step gave a birth to the HCI paradigm.

According to Kuhn, the new HCI paradigm replaces the old one during the *"Paradigm Change"* step. This replacement didn't happen in the case of HCI. The emergence of HCI created a new Kuhn Cycle for HCI as depicted in figure 10.2.

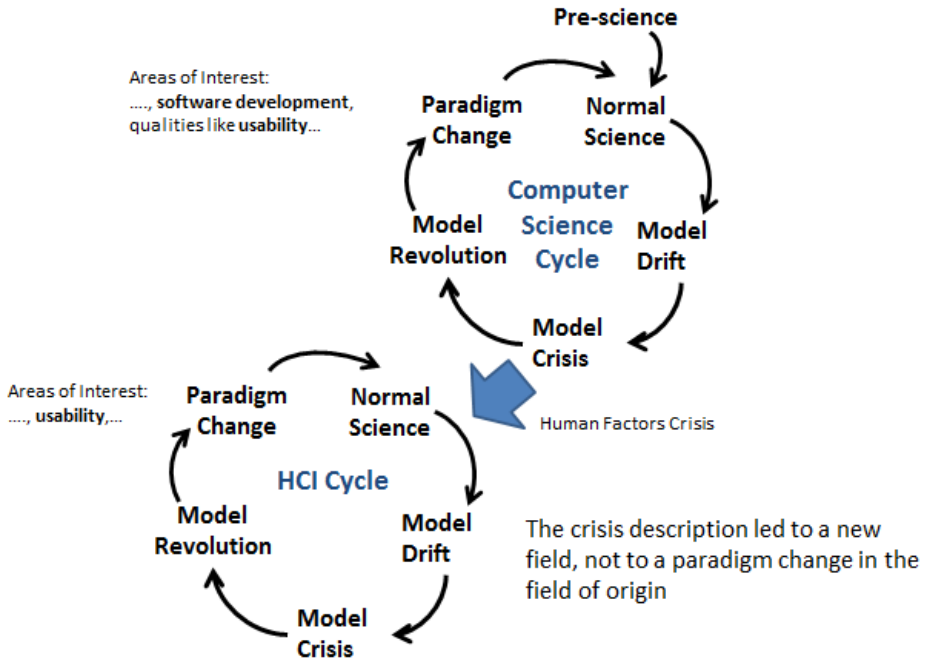

Figure 10.2 The creation of the HCI cycle after the Human Factors crisis.

HCI and SE evolve in different cycles, with different areas of interest. **HCI focuses on tasks of people using a system or product**, infor-

mation and interaction requirements to carry out these tasks, in a work environment. This focus designates the area of interest in the HCI Cycle. On the other side, **SE focuses on software capabilities required for the system to perform its functions, achieve objectives, and satisfy requirements**. However, HCI and SE share the interest in mainly usability.

With the creation of a new HCI cycle, accompanied with a shift in the area of interest, the HCI discipline could get the privilege to address usability, while SE kept the privilege to address the development of software products. This shift in the area of research carried out new opportunities to HCI by joining the efforts and expertise in different disciplines towards the usability as the common objective.

This shift in the area of interest of the HCI cycle carried out limitations to the HCI. HCI cannot proclaim the ability to develop software products because it does not have the required tools[37] to analyze, design and develop a full software product. Therefore, HCI became dependent on SE to apply its undeniable usability findings in real software products. Furthermore, because SE has the privilege to address the development of software products, it has the right to judge on HCI contributions as useful for the software development or not. HCI researchers find themselves in a position to defend and demonstrate the benefit of their findings to the software development, which is beyond their main interest which is defined in the HCI discipline as *usability*. Due to this limitation, HCI tends to proof its findings on prototypes most often.

The limitations imposed on HCI due to the scientific evolution path described using Kuhn Cycle manifest in another form. Researchers in SE and HCI have had long debates on who should lead the software development, the SE or the HCI. Interested readers can read more on this debate in [Buie97]. From our point of view, this long debate is because researchers from both disciplines are not able to communicate and understand each other. This inability to communicate is predicted by Kuhn during the *"Model Revolution"* phase: *"Believers in each paradigm cannot communicate well"*.

An illustrative example on the inability to communicate between SE and

[37] Required tools include tools to gather functional/non-functional requirements, effective design and development tools.

HCI researchers is the task model. The task model is appreciated by HCI researchers, while it is almost ignored by SE researchers. We think the discussion of the evolution of HCI using the Kuhn Cycle of the evolution of science may provide an explanation of why this is happening. The *"Model Revolution"* and the creation of a HCI cycle is the reason why the communication between both communities is difficult.

10.3 Towards a Linguistic Paradigm

Imagine that the new model of understanding introduced by HCI led to a model revolution in the Computer Science cycle and did not lead to the creation of a new cycle. The Computer Science would have evolved in the light of the new HCI model of understanding. The new model of understanding would have replaced the old one leading to the *"Paradigm Change"* step. This would have happened if the HCI model proved to be successful to address the human factors crisis which is the main motivation behind the existence of HCI at the first place.

The evolution of a discipline depends on the understanding of the problems that confront the discipline. The understanding of the Human Factors crisis guided the evolution path, as HCI and Computer Science provided different interpretations to the crisis. The Human Factors crisis is stated as:

Human factors are positioned at the late detailed design phase of the waterfall: the design of the User Interface (UI). Thus, it is involved after main design decisions had been taken. It is limited to cosmetic differences in software products.

HCI interpreted the crisis description as: the crisis happens because human factors are positioned at the late design phase. The solution is: move human factors to the early analysis phase. The means to do it is: by employing cognitive models to analyze the human mind and behavior when using the computer. The evolution path that resulted from this interpretation led to the evolution of HCI integration models that we discussed in chapter 2.

Computer Science interpreted the crisis description as: The crisis happens because we cannot get the end-user's feedback on the UI (and on the product in general). The solution is: to get the end-user's feedback as fast as possible. The means to do it is: iterative development methods. The evolution path that resulted from this interpretation led to

the evolution of prototyping approaches, iterative and incremental SDLCs and to agile methods.

However, there is a third interpretation that was never explored before: the crisis happens because the UI design is at the late design phase (which forces addressing human factors at the late design phase). The solution is: start the UI development since the analysis phase. The means to do it is: by looking for a new perspective to the UI development that perceives the UI at the analysis phase.

This thesis followed this third interpretation. It developed the linguistic perspective that allows developing the GUI since the analysis phase. It proved that this linguistic perspective can be used to develop a GUI following a Model-Based approach (the case study on search for a flight).

The linguistic perspective introduces a new model of thinking on how to develop a GUI. Therefore it moves the Computer Science cycle to the *"Model Revolution"* step.

We argue that the linguistic perspective has the potentials to move the Kuhn Cycle towards the *"Paradigm Change"* step, because:

1) The linguistic perspective provides a comprehensive model of understanding of the GUI development, from the top task level to the lowest widgets properties level.
2) The linguistic perspective is a new perspective to the development of the GUI, based on a classification-based[38] and refined approach. It has a well-defined modeling framework on different levels of abstractions with well-defined interfacing between these levels[39].
3) It informs computer scientists how to address problems in the GUI by firstly classifying the problem's impact on the GUI then implement the impact per-level. Examples include the problem of integrating usability guidelines in the development (discussed in section 4.4.2) or the problem of adapting the GUI to different contexts of use (discussed in section 4.4.3).
4) It impacts how to do abstraction of GUI concepts by employing an explicit classification system: the linguistic classification

[38] The Linguistic Prism depicted in figure 4.8

[39] The Linguistic GUI modeling framework depicted in figure 4.10.

5) It has implications on how to do modeling. The model slice[40] is a concept that is introduced in the linguistic modeling approach. Besides, the linguistic modeling approach defines a set of requirements (identified in section 4.5) for a model to be considered as a linguistic model.

The above characteristics of the linguistic perspective are aligned with Kuhn definition of a paradigm: *a paradigm is a comprehensive model of understanding that provides field's members with viewpoints (perspectives) and rules on how to look at the field's problems and how to solve them*. Based on these characteristics, we argue that the linguistic perspective has the potentials to move the Computer Science to the *"Paradigm Change"* step.

The paradigm change in the Computer Science cycle will definitely impact the HCI cycle because Computer Science is part of the fields constituting HCI. More concretely, the linguistic perspective and the presented linguistic GUI model in this thesis are developed from HCI ideas: the task model and the GUI development on multiple levels. Chapter 9 demonstrated how the linguistic perspective can be integrated in a SDLC. Therefore, even HCI from the paradigm shift in the Computer Science because it can open new opportunities to allow researchers from SE and HCI to better communicate. HCI may also become less dependent on SE because the linguistic development can start from any level and integrated later in the final product.

In the coming sections, we present different examples on new opportunities created by the linguistic perspective. The purpose of these examples is to increase the common sense of the paradigm shift that might happen based on the linguistic perspective.

10.4 The Paradigm Shift in Agile Methods

Chapter 9 demonstrated how to integrate the linguistic perspective in a SDLC. In this section, we explain that the linguistic perspective can be integrated in different methods of software development, with no limitations on the targeted method. The reason behind this flexibility is because the linguistic perspective introduces a way of thinking and not a development method. The method is defined to solve a problem based

[40] The part of the model on a level.

on the linguistic perspective.

We have already discussed Scrum in chapter 2. The reader may refer to section 2.7.3 for more information on agile methods. However, the linguistically-adapted Scrum framework is depicted in figure 10.3. The adaptation concerns two parts: the product backlog, and the sprint.

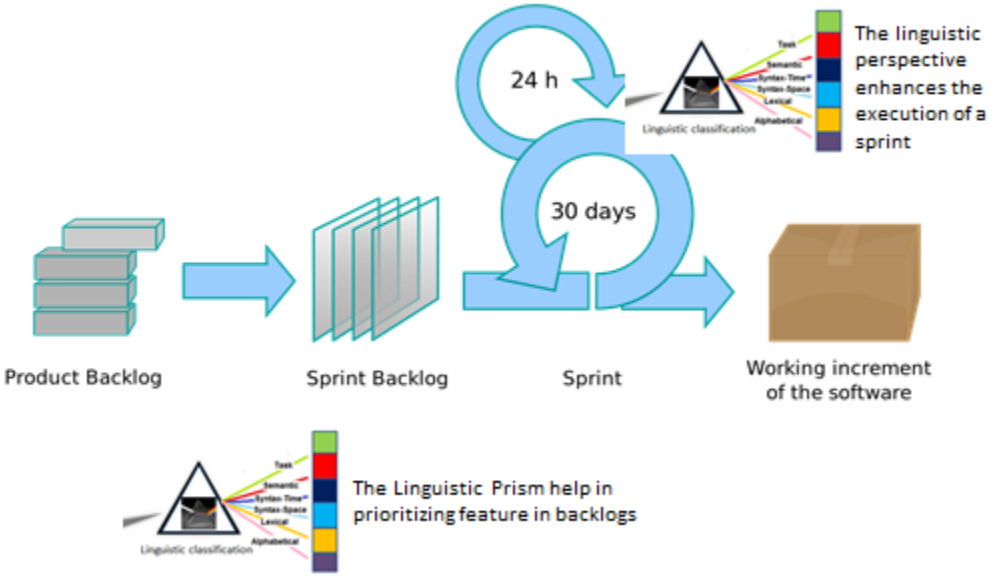

Figure 10.3 The Scrum framework with impact points of the linguistic perspective identified.

10.4.1 Adaptations to the Backlog

The Scrum backlog is a prioritized list of features that is maintained by the product owner. Features are created from the list of activities to develop the product through a process called grooming. At the beginning of each sprint, the team selects a set of features from the backlog it believes it can complete. This selection activity is called the sprint planning and it forms the sprint backlog.

The Linguistic Prism (the classification tool depicted in figure 4.8) can help in estimating the work required for GUI-related features both in the product backlog and in the sprint backlog. The GUI-related estimation is based on classification of the feature impact on the GUI. A feature that

impacts the task level requires more work than a feature that impacts the semantic level because the change on the task model may propagate to all lower levels. Besides, an estimation of the effort per- level is also possible based on the change required. For instance, adding three tasks to the task model requires more effort than simply modifying the relation between two tasks.

A numeric estimation of the required GUI effort can be deduced. If a feature requires a modification on a level, we give it the value 1, if not we give it a value 0. If we map the linguistic levels to a 6-bits binary numeric value, where the left-most bit depicts the estimate of required GUI effort on the task level while the right most bit depicts the estimate of required effort on the widgets-properties level, we can have a binary representation of the estimated GUI effort per-feature. This is depicted in figure 10.4. This estimation can be extended to use a hexadecimal system thus enabling a refined estimation per-level from 0 to 15.

10.4.2 Adaptation to the Sprint

In a Scrum sprint, all the activities necessary to create the product increment are performed. This includes: analysis, design, integration, building and testing. At the end of a sprint, when we have the working product, we can get the feedback of the user (or the user representative role) then adapt the increment based on this feedback. This is depicted in figure 10.5. Recall that we identified a limitation in agile methods concerning usability testing in chapter 2. The limitation is that testing the usability of iteration *i* is made in iteration i+1 and considered in iteration i+2, which creates a phasing problem. Besides, the UI designer was given an iteration ahead to start with the design in the iteration i-1.

The linguistic perspective can enhance this process on three issues:

1- Better organization of UI activities: the first step to develop a feature from the sprint backlog is the analysis activity. The linguistic GUI can start with the analysis phase to ensure usability requirements thanks to the task model used at the task level (refer to section 9.2.1). When designing the feature, the linguistic GUI can also participate in the design to ensure usability requirements consideration in this phase (refer to section 9.2.2). A working Semantic-GUI can be consulted at the end of this phase. After the design activity, the GUI development can be conduct-

ed in parallel with the development of other modules for the feature.

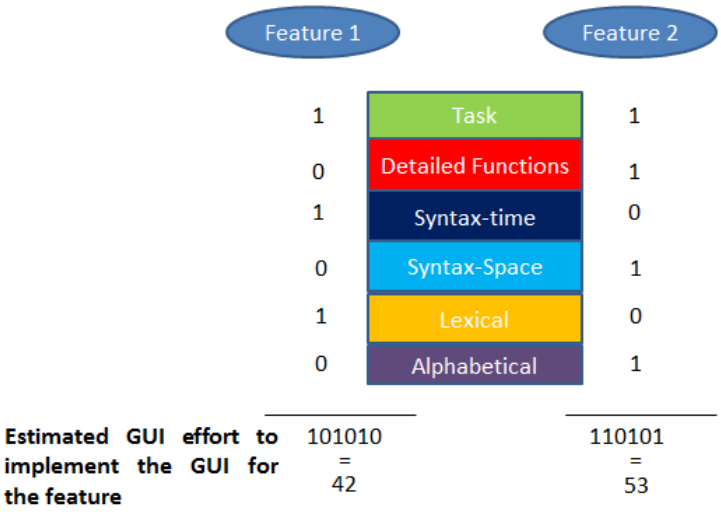

Figure 10.4 A linguistic estimation method for the effort to implement the GUI for a feature.

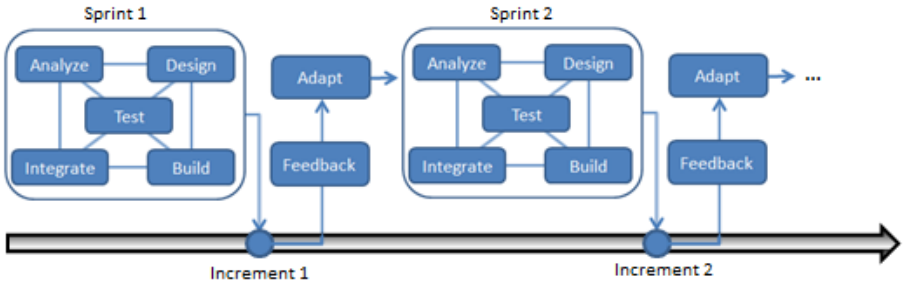

Figure 10.5 Sprints in scrum and activities enacted inside a sprint.

2- The feedback can be integrated per-feature during the sprint and not per-increment. The linguistic GUI enables getting the feedback of the user (or the user representative role) during the sprint thanks to the concrete GUI per-level. This enables different possibilities to get the feedback: (1) during the feature development (2) after the feature development, in addition to (3) per-increment. This in turn creates the possibility to start the adapta-

tion (based on the feedback) during the iteration itself. However, the feedback and adapt activities after the increment are still necessary, but they might be reduced dramatically with the feedback-per feature.

3- The phasing problem in Scrum is radically minimized because the usability testing can be performed alongside the sprint development or even the feature development. Note that if usability test are carried out per-feature during the sprint execution (refer to figure 9.4), the feedback that is focused on aesthetics because usability testing on the linguistic GUI at high-levels have already be performed. Therefore, the phasing problem is almost voided.

Figure 10.6 illustrates how the sprint activity can be executed when integrating the linguistic perspective within. We call adapted sprint to the linguistic perspective a Linguistic Sprint to highlight the linguistic perspective within.

10.5 Operationalization of Usability Guidelines

Operationalization of usability guidelines aims at integrating usability guidelines automatically in the UI development. Operationalization efforts succeeded on design rules (a type of usability guidelines) because they are concrete and clear enough to developers.

The linguistic perspective provides new opportunities to the operationalization of guidelines. This is because it allows the development of a GUI at any level, without the need to start from the top level or even have anything on lower levels. This section demonstrates this method of GUI development. We will use the proposed linguistic model in this thesis in the demonstration.

The main characteristics of the linguistic perspective that is the key enabler of new guidelines operationalization opportunities is the *univocal repartitioning of GUI concepts on linguistic levels*. The key enabler in our linguistic model is the use of tags to define relations among UI elements dynamically (refer to figure 6.2).

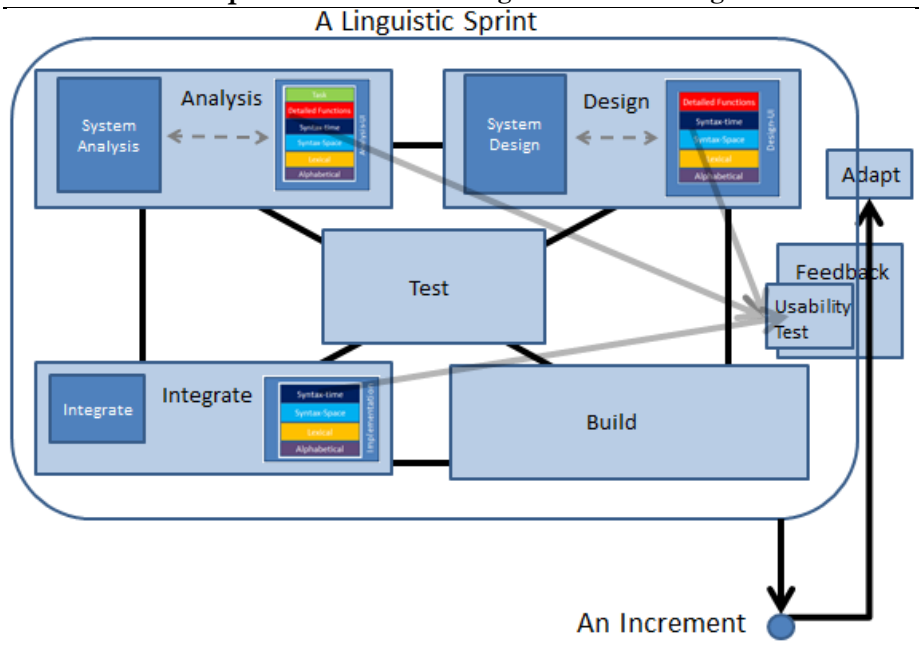

Figure 10.6 The Linguistic Sprint

The linguistic classification of a usability guideline identifies the GUI concepts that are impacted by it. Therefore, it identifies the linguistic level and the part of the linguistic model that is affected. Examples on how to classify guidelines are presented in section 4.4.2. However, we present some examples here.

Before demonstrating how to operationalize this guideline, we need first to explain how operationalization is performed in the linguistic model. Operationalization of guidelines takes the form of <*UI Patch, Glue*>. The *Glue* is the code[41] fragment that identifies the group of elements on which to apply the UI Patch. A *UI Patch* is the Prism programming language code that is applied on elements identified in the glue. The glue is a code that is written when we want to apply the guideline in a specific application. The UI Patch can be written during the classification of the guideline, therefore we can think of a catalog of usability guidelines that contains UI Patches already programmed for each guideline (a library of

[41] In this chapter, the word "code" always refers to a code that is written in the Prism programming language.

UI Patches).

10.5.1 Operationalization on the alphabetical level

Assume a company defines the following design rule:

Critical tasks should be displayed with a red foreground to attract the user's attention.

The guideline we want to operationalize impacts the background color attribute of widgets. Therefore, based on table 3.2, this guideline is classified[42] at the widgets properties linguistic level. The UI Patch for this guideline is:

```
CriticalTaskSytlistics (UIElementCollection selectedElements){
        //This code is in the Prism programming language
        alph::select e from selectedElements where true {
                alph::e.uiWidget.ForegroundColor= "red" ;
        }
}
```

The UI Patch has a given name and a parameter that represents a list of UI elements. The glue provides this list of UI elements to the UI Patch when applied in a specific application. A glue for an application would look like the following:

```
CriticalTaskSytlistics (select e from uiElements where e.tags.task = "criticalTaskName");
```

In programming languages terms, the glue is simply a call to the guideline procedure while passing the list of UI elements to apply the guideline on.

The company can develop the UI Patch once, and customize the glue for every application it develops. Customization of the glue can be enforced at the time of integrating the guideline in the application.

10.5.2 Operationalization on the Syntax-space Level

Another guideline that is widely used in GUIs is:

---

[42] One of the characteristics of the linguistic model (inherited from the linguistic perspective) is that if the developer wants to manipulate a concept, there is only one level to do it: the level that defines the concept. For instance, the color concept can be only modified on the widgets properties level. Therefore, the developer can tell directly the classification of a guideline by coding the UI Patch.

On a window for editing, align labels to the right and text boxes to the left.

The UI Patch becomes contains statements on the syntax-space level. It takes the form:

```
LabelBeforeTextbox (UIElement label, UIElement textbox){
        //This code is in the Prism programming language
        syns::label before textbox;
}
```

The glue is required to identify the elements to apply the guideline on. It is responsible to identify the UI elements in a window for editing to apply the guideline on. However, the glue can benefit from the high traceability in the linguistic model. For instance, if a task's goal is to edit some information, the glue can select all UI elements that belong to this task. If a detailed function is responsible for editing, the glue can select UI elements that belong to this detailed function. Other possibilities like selecting/excluding specific UI elements are also available depending on the case in hand.

10.5.3 Operationalization on the Syntax-time Level

Take the usability guideline from [Vanderdonckt2007]:

To display principal and secondary information belonging to the same task, use an extensible dialog box.

An example on this guideline is depicted in figure 10.7. This GUI allows to replace some information in Microsoft Word. The UI elements for principal information to perform the replacement are displayed on the left side. When pressing the "More>>" button the UI elements for secondary information are displayed as in the right side of the figure.

This guideline impacts the navigation concept, which is at the syntax-time level. It depicts two time containers principal and secondary. The principal time container contains UI elements for principal information while the secondary time container contains UI elements for secondary information. The relation between both containers is of type browsable. To create the guideline, we have to create the two time containers and the relation between them. The UI Patch is:

```
PrincipalSecondaryNavigation (TimeContainer parentContainer, UIEle-
mentCollection principal, UIElementCollection secondary){
        //This code is in the Prism programming language
```

```
        synt::TimeContainer principalContainer;
        synt::select e from principal where true
{e.timeContainer=principalContainer;}
        synt::TimeContainer secondaryContainer;
        synt::select e from principal where true
{e.timeContainer=secondaryContainer;}
        synt::browse secondaryContainer from principalContainer;
        synt::observe principalContainer from parentContainer;
}
```

The glue will take care of identifying what UI elements are considered as principal and what information can be considered as secondary, according to the application implementing the guideline. The glue can benefit from the high traceability in the linguistic model to identify these elements.

10.5.4 The Operationalization Method

The examples above explain how the operationalization of usability guidelines can be performed on the linguistic model. The main benefit that the heart of usability embedded in the guideline is extracted through the linguistic classification by identifying impacted GUI concepts. A UI Patch can be developed at the time of classification creating a library of code fragments for each usability guideline.

To apply a guideline in a specific application, we need a glue to extend the application with the UI Patch. This glue is responsible to interpret the conditions of application of the guideline in the application. The glue can benefit from the high traceability of concepts in the linguistic model.

The method to operationalize a guideline in the linguistic model is depicted in figure 10.8. We call this method: the linguistic usability guidelines operationalization method.

10.6 Adaptation of the GUI

The linguistic model is independent from the context of use. Therefore, it is up to the modeler to employ desired model for context of use among other HCI models. The key support for the adaptation in the linguistic model is the support for the evolution of the GUI (see section 4.4.3).

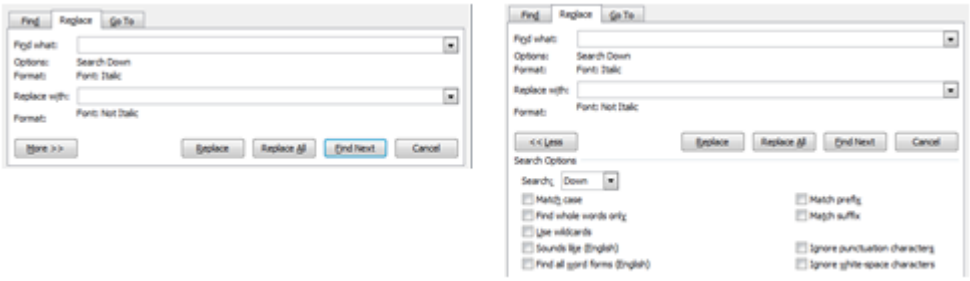

Figure 10.7 An example on a usability guideline implementation.

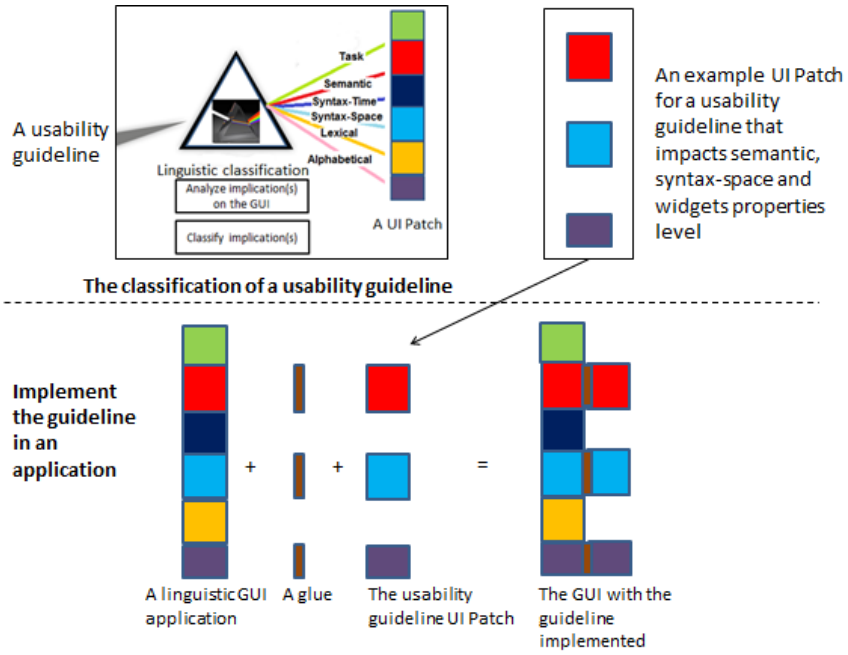

Figure 10.8 The linguistic usability guidelines operationalization method.

We demonstrate how to do adaptation on a simple example. Assume a GUI that is already developed using the linguistic model. We need to adapt this GUI to the context of use that is defined (for simplicity) on two dimensions: the level of experience of the user (novice, expert) and the size of the screen (small, large). This context of use depicts 4 usage scenarios that should be supported in the GUI: (1) a novice user using a small screen, (2) a novice user using a large screen, (3) an expert user us-

ing a small screen and (4) an expert user using a large screen).

The first step in the adaptation of the GUI is to linguistically classify the impact of each of the usage scenarios on the GUI. The result of this classification is a UI Patch for each usage scenario. However, UI Patches can be created atop of other ones. For instance, if it appears that the impact of small screen on the GUI is the same for expert and novice users, we can extend the UI Patch for the usage context *"a novice user using a small screen"* from the UI Patch for the usage context *"expert user using a small screen"* instead of extending it from the initial version of the GUI. This is depicted in figure 10.9.

The running GUI needs to detect the change in the context. For this purpose, it employs context change sensors which can be any possible means to detect such a change (a hardware device to detect changes in the environment, a profile for the user, an explicit action from the user to change the context, etc.). The detected change is passed to the linguistic router which has a routing table that informs it on the sequence of applying UI patches to adapt to the context of use. The routing table can be modeled as rules (condition action).

The creation of a UI patch of every usage scenario is not mandatory. The routing table in the linguistic router can be modeled to determine the appropriate UI Patches to apply and their sequencing.

The independence of the linguistic model from the context of use model enables evolving[43] the context of use model without breaking the GUI. This can be supported by versioning the routing tables and mapping each version with the version of the context of use. As a result, the GUI can be used at the same time for two contexts of use models.

[43] This is potentially to happen because context of use models are HCI models that evolve over the time.

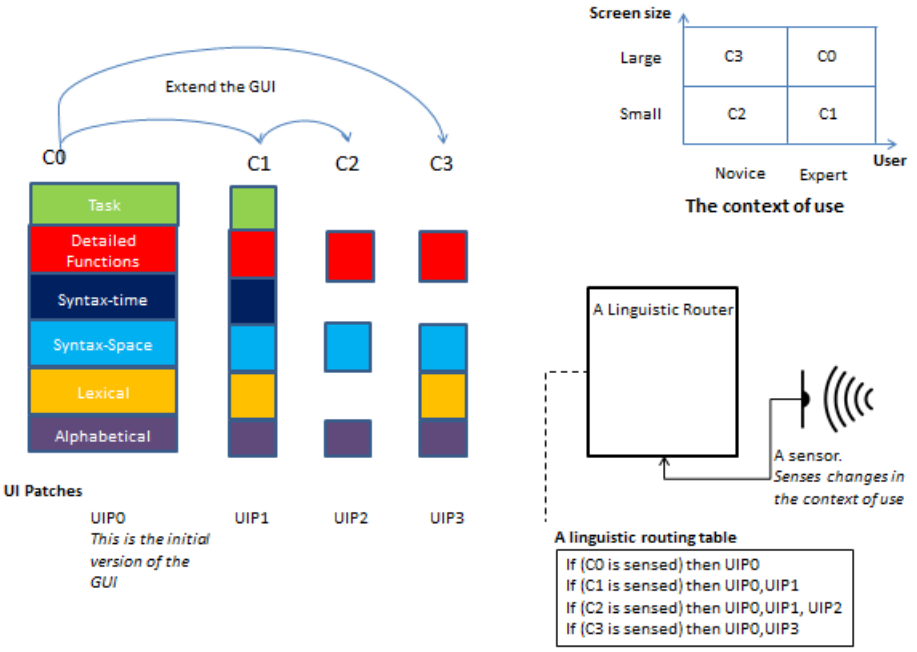

Figure 10.9 An example on adapting a GUI to a context of use

Chapter 11 Conclusion

11.1 Context

Since the identification of the human factors crisis, efforts to integrate usability in the software development have not stopped. Facing the fact that this is a long lasting problem, we tried to address it from a different perspective, with the aim to provide a different point of view and a different understanding of the problem.

This dissertation is a step in the direction of sensing the problem from a non-traditional way of thinking in both the HCI and Software Engineering. The problem is immune to all previous efforts. Therefore, it should have deep roots to reveal, understand and address.

Although we may share this very same idea with many other researchers, we tried to make a practical step on a one thousand miles road. This dissertation tries to stimulate critical thinking and different perspectives to the problem.

We also believe that HCI cannot solve the problem alone, neither can Software Engineering. A need for more collaboration is needed, with a focus on the software product itself.

This chapter summarizes contributions brought in this dissertation to the domains analyzed and addressed in the state of the art: HCI and Software Engineering. We classify our contributions on three levels:

1) Conceptual contributions
2) Frameworks and specification models
3) Implementation

11.2 Summary of contributions

Below is a list of main contributions of this thesis.

11.2.1 On the conceptual level:

- **C1**. A Linguistic Software Development life cycle: the prism-

SDLC. A SDLC that aligns the GUI development with software development activities [Khaddam2016].

- **C2**. A linguistic Perspective to the GUI development: a perspective to the GUI that allows building a working GUI since the analysis phase. This perspective is based on considering the interaction between the user and the machine as a written text, that can be analyzed linguistically [Khaddam2015a].
- **C3**. A linguistic Taxonomy of the interaction: a theoretical study of systematic linguistic classification of GUI concepts, artifacts and activities, including their bases, principles, procedures, and rules [Khaddam2015a].

11.2.2 On the frameworks and specification models level

- **C4**. A linguistic classification of requirements: as one outcome of the linguistic taxonomy. This classification is used to classify changes on the GUI on linguistic levels [Khaddam2015a].
- **C5**. A Linguistic classification of GUI concepts and activities: It allows to univocally separate GUI concepts and activities per linguistic levels [Khaddam2015a].
- **C6**. A Linguistic Modeling Framework: that explains how to start from requirements and integrate them at the right development level in the GUI development process [Khaddam2015b].
- **C7**. GUI Linguistic Modeling Requirements: We elicited a set of requirements to establish a GUI linguistic model. These requirements are organized by linguistic level [Khaddam2015b]. These requirements are essential for a model to be considered as a linguistic model.
- **C8.** A set of linguistic models for the GUI: A proof-of-concept linguistic model that fulfills requirements in C7. This includes the linguistic task model [Khaddam2015c].

11.2.3 On the implementation level

- **C9.** The Prism programming language: A linguistic GUI development language that is used to develop linguistic GUIs.
- **C10.** A case study implemented using the prism-SDLC following a linguistic model-based approach. It is a proof of concept that a working linguistic GUI is realistic.

These contributions are depicted in the pyramid of contributions in figure 11.1.

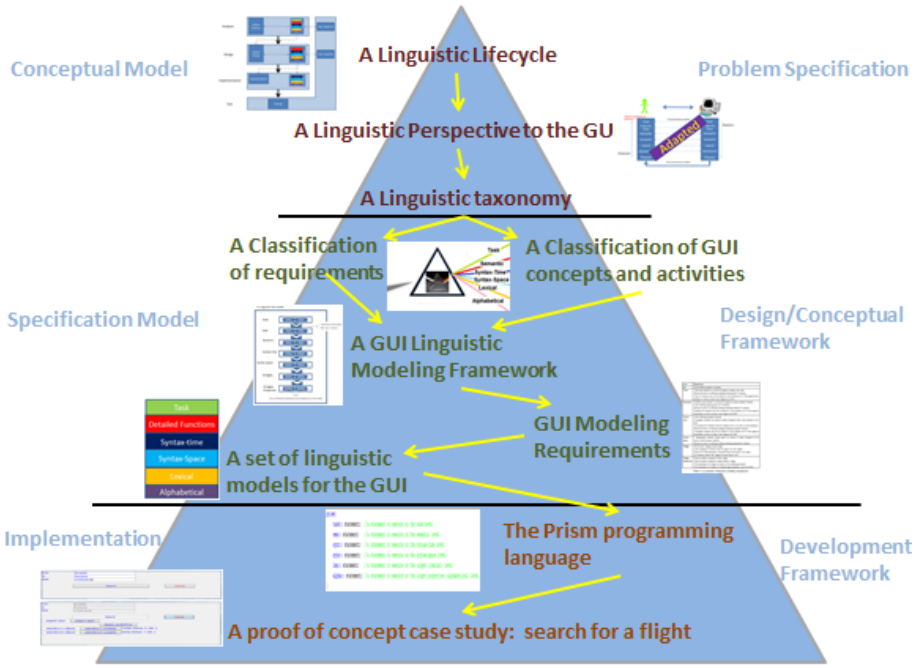

Figure 11.1 The pyramid of contributions of this thesis

11.3 Validation

The solution process passed through three milestones. Validation in each milestone is different from others.

In the first milestone, we validated the linguistic perspective on a case

study. We explained how a GUI for a "register for a conference" could be established.

In the second milestone, we provided a proof of concept of the ability to develop a working GUI on a simple case study: search for a flight. Although the case study is simple, it fits the purpose. A more complicated case study cannot be explained at the level of detail required in the proof of concept.

In the third milestone, we had realistic constraints on the feasibility of validating a development lifecycle. We explained how the Prism-DLC is induced from the structured GUI development established from the linguistic perspective, which conforms to the thesis statement.

11.4 Scope

Although our discussions were focused on GUIs, we think it is possible to extrapolate the findings and contributions on other types of UI. The linguistic meta-model is capable to be extended to the specific needs of other types of UIs, because the only levels that are GUI-specific are the widgets and widgets properties levels. Upper levels are independent of the UI type. However, this should not be interpreted as upper levels are platform-independent or modality-independent because even the task model might require adaptation to the type of modality or platform. However, to support different types of UIs, we should review meta-models in upper levels to fulfill specific requirements for a UI type.

11.5 Limitations

Our validation focused on the feasibility of the linguistic perspective. It provides evidence that the perspective can be applied in real GUI development of a software product.

A main limitation in this dissertation concerns validating the Prism-DLC. With no practical evidence, it is hard to claim the promises of the Prism-DLC. This limitation is due to two obstacles:

1) Lack of appropriate designer tools for models on levels.
2) Lack of resources to validate the development lifecycle.

Besides, proposed linguistic models are not intended to be complete. Their capabilities, among other qualities, were not part of the scope of

this dissertation. This is also a limitation to conduct further research on the linguistic modelling approach.

Although we provided a technical environment to develop linguistic GUIs, by introducing the Prism programming language, there are many aspects that should be considered before enabling developers to use it. Some of these aspects are related to appropriate development environment, enhanced syntax, enhanced compiler and error checking among others. This limitation prevents the use of the Prism programming language from being used in practical products.

11.6 Future work

As we said before in this chapter, we believe this is a step on a long road. There are various tasks on both short-term and long-term perspectives.

The most important work to do in the future is to establish the new paradigm to the GUI development that we discussed and characterized in chapter 10 and to demonstrate, validate and prove its promises and benefits. The works described in the coming sections are mandatory towards a new paradigm.

11.6.1 In a short-term perspective

First, we need to ensure modelling qualities in our proposed models on each level. The capability quality is one of the most important to ensure due to its impact on the produced GUIs.

Second, we need to develop graphical designer tools, at least for the two upper levels: the task and the semantic levels. The importance of these levels is mainly related to the roles intended to use them: analysts and designers. These two roles prefer graphical tools rather than programming languages. Other lower levels are also important especially if we want the tools to be used side by side with the client to test usability and/or to gather UI requirements from the user.

Third, we need also to enhance the Prism programming language to enhance its capabilities and its tool support by developing appropriate tools.

Finally, when all the above is performed, we can open the possibilities to validate the Prism-DLC in real applications. This validation is critical to

provide real measures on the performance of the development lifecycle and achieving its promises.

11.6.2 In a long-term perspective

It looks to us that the linguistic perspective opens more questions than it answers. Examples of these open questions are:

In HCI:

1) How operationalization of usability guidelines can be addressed from a linguistic perspective? The linguistic classification is promising but more research is required to evaluate consequences on the operationalization. The operationalization of usability guidelines is the automatic incorporation of usability guidelines in the UI development. The discussions in chapter 10 should be validated on real applications.
2) Can the linguistic perspective be extended to other UI modalities than the GUI (vocal, haptic, virtual reality, etc.).
3) How to support different contexts of use? We argued in this dissertation that the linguistic perspective changes the role of HCI models to support the UI development, instead of leading the development. This change in the role has consequences to investigate on how to model them and how to incorporate them in a linguistic modelling approach. Recall that the linguistic modelling approach provides support to such models through extensions.
4) What is the impact on other concepts in HCI? All concepts in HCI are defined from the common perspective to the UI development. How these concepts are impacted from the linguistic perspective? Examples of these concepts are: adaptable/adaptive UIs, low/high fidelity UIs, context-aware UIs among others.

In Software Engineering:

5) What are the implications of the linguistic perspective on software qualities in the developed UI? We discussed the impact on traceability. But what are other qualities impacted? Maintainability is a good candidate to start with.
6) Is it possible to apply the linguistic perspective on the development of other modules than the UI in a software product? The overall software engineering can profit from characteristics of the

linguistic perspective.

7) How Model-Driven Architecture (MDA) and Model-Driven Engineering (MDE) can be impacted by the linguistic perspective? Especially if it is extended to the complete software development. The linguistic perspective provides a new way of abstraction based on an explicit classification. How this new way of abstraction affects modelling approaches? To what extent it can be applied in other fields?

With the linguistic perspective, new avenues are opened. We hope it is the start towards new and innovative perspectives to the UI development, which might get HCI and Software Engineering even closer.

Chapter 11. Conclusion

My Publications

Khaddam, I., & Vanderdonckt, J. (20 June 2010). Adapting UsiXML User In-terfaces to Cultural Background. 1st Int. Workshop on User Interface eXtensible Markup Language UsiXML'2010 (pp. 163–170). Berlin: Thales Research and Technology France, Paris (2010).

Khaddam, I., & Vanderdonckt, J. Flippable User Interfaces for Internationalization. EICS 2011, June 13–16, 2011, Pisa, Italy.

Towards a Culture-Adaptable User-Interface Architecture. Revista Romana de Interactiune Om – Calculator. Vol.7, Num 2. Special Issue: User interface standardization 2014. ISSN 1843-4460 (pp. 161 – 194).

Khaddam, I., Mezhoudi, N., Vanderdonckt , J. A Linguistic Perspective to Develop Graphical User Interfaces. 3ed International Conference on Control, Engineering & Information Technology (CEIT). Tlemcen, Algeria, 25-27 May, 2015.

Khaddam, I., Mezhoudi, N., Vanderdonckt , J. Towards a Linguistic Modeling of Graphical User Interfaces: Eliciting Modeling Requirements. 3ed International Conference on Control, Engineering & Information Technology (CEIT). Tlemcen, Algeria, 25-27 May, 2015.

Khaddam, N. Mezhoudi and J. Vanderdonckt, "Adapt-first: A MDE transformation approach for supporting user interface adaptation," Web Applications and Networking (WSWAN), 2015 2nd World Symposium on, Sousse, 2015, pp. 1-9. doi: 10.1109/WSWAN.2015.7209080

Khaddam, I., Mezhoudi, N., Vanderdonckt , J. Towards Task-Based Linguistic Modeling for designing GUIs. 27ème conference francophone sur l'Interaction Homme-Machine. Oct 2015, Toulouse, France. ACM, IHM-2015, pp.a17, 2015

Khaddam, I., Barakat, H., Vanderdonckt , J. Enactment of User Interface Development Methods in Software Life Cycles. Proceedings of 13th Int. Conf. on Human-Computer Interaction ROCHI'16, September 8–9, 2016, Iasi, Romania. MATRIX ROM, Bucharest, 2016, pp. 26-35. ISSN 2501-9422.

My Publication

Khaddam, I., Bouzit, S., Calvary, G., Chêne, D., MenuErgo: Conception assistée de menus par évaluation automatique de règles ergonomiques. Proc. of AFIHM Int. Conf. on Human-Computer Interaction IHM'16, October 25–28, 2016, Fribourg, Switzerland, ACM Press, New York, 2016, pp. ? ISBN 978-1-4503-4322-0.

Co-Author

Mezhoudi, N., Khaddam, I., Vanderdonckt , J., Toward Usable Intelligent User Interface. International Conference on Human-Computer Interaction. (pp. 459-471). Springer International Publishing. 2015.

Mezhoudi, N., Khaddam, I., Vanderdonckt , J., WiSel: a mixed initiative approach for widget selection. Proceedings of the 2015 Conference on research in adaptive and convergent systems. (pp. 349-356). 2015.

Mezhoudi N., Perez Medina JL., Khaddam I., Context-Awareness Metamodel for User Interface Runtime Adaptation. International Journal of Software Engineering. Volume 2. 2015.

Mezhoudi N., Perez Medina JL., Khaddam I., Vanderdonckt , J., Towards a Conceptual Model for UIs Context-Aware Adaptation. 2nd World Congress on Computer Applications and Information Systems WCCAIS'2015.

References

A

[ACM1994]

Hewett, TT., Baecker, R., Card, S., Carey, T., Gasen, J., Mantei, M., Perlman, G., Strong, G., Verplank, W., 1992. ACM SIGCHI Curricula for Human-Computer Interaction. ACM New York, NY, USA. ISBN:0-89791-474-0.

[Annett1967]

Annett, J., Duncan, K.D., 1967. Task analysis and training design. Occupational Psychology, 41, p211-221.

[Annett1971]

Annett, J., Duncan, K.D., Stammers, R.B., Gray, M.J., 1971. Task Analysis. Department of Employment Training Information paper 6. HMSO, London.

B

[Banks1983]

W.W. Banks, W.E. Gilmore, H.S. Blackman, D.I. Gertman, Human Engineering Design Considerations for Cathode Ray Tube-Generated Displays, Vol. II, Document NUREG CR-3003/EGG-2230, U.S. Dept. of Energy, Idaho National Engineering Laboratory, Idaho Falls, Idaho, July 1983.

[Bannon1991]

Bannon, L., Bodker, S. Beyond the interface: Encountering artifacts in use. In J. M. Carroll (Ed.). Designing interaction (pp. 227–253). Cambridge, UK: Cambridge University Press. 1991.

[Barboni2010]

Barboni, E., Ladry, J., Navarre, D., Palanque, P., Winckler, M.: Beyond modelling: an integrated environment supporting co-execution of tasks and systems models. In: Proceedings of the 2nd ACM SIGCHI Symposium on Engineering interactive Computing Systems. EICS 2010, pp. 165–174. ACM, New York (2010).

References

[Barthet1996]
Barthet, M.-F., & Tarby, J.-C. (1996). The Diane+ method. In J. Vanderdonckt, (Ed.), Computer-aided design of user interfaces (pp. 95–120). chapter 26 Namur, Belgium: Presses Universitaires de Namur.

[Beck2004]
Beck K., Andres, C. (2004) eXtreme Programming Explained: Embrace Change, second edition. Addison-Wesley. ISBN: 0321278658.

[Betri2004]
Berti, S., Correani, F., Paterno, F., Santoro, C., The TERESA XML language for the Description of Interactive Systems at Multiple Levels. In: Proc. Of the Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages, pp. 103-110 (2004).

[Beyer2010]
Beyer, H. User-Centered Agile Methods. Ed. John M. Carroll. Morgan & Claypool Publishers series synthesis lectures on human-centered informatics. ISBN: 9781608453726. DOI 10.2200/S00286ED1V01Y201002HCI010.

[Bezivin2001]
Bézivin, J., Gerbé, O. (2001) "Towards a Precise Definition of the OMG/MDA Framework", ASE'01, November 2001.

[Bodart1996]
Bodart F., Vanderdonckt J. Widget Standardisation through Abstract Interaction Objects (1996). In « Advances in Applied Ergonomics », Proceedings of 1st International Conference on Applied Ergonomics ICAE'96 (Istanbul, 21-24 May 1996), A.F. Özok & G. Salvendy (Eds.), USA Publishing, Istanbul - West Lafayette, 1996, pp. 300-305

[Boehm2000]
B.Boehm. (2000) Requirements that handle IKIWISI, COTS, and Rapid Change. Computer, IEEE, 34(5), pp. 99-102.

[Buie97]
Buie, E., A., & Vallone, A. (1997). Integrating HCI engineering with software engineering: A call to a larger vision. In Smith, M. J., Salvendy, G., & Koubek, R. J. (Eds.),Design of Computing Systems: Social and Ergonomic Considerations (Proceedings of the Seventh International Conference on Human-Computer Interaction), Volume 2. Amsterdam, the Netherlands: Elsevier Science Publishers, 1997, pp. 525-530.

C

[Caffiau2010]
Caffiau, S., Scapin D., Girard, P., Baron, M., Jambon, F. (2010) Increasing the expressive power of task analysis: Systematic comparison and empirical assessment of tool-supported task models. Interacting with Computers. Volume 22 Issue 6, November, pp. 569-593. Elsevier Science Inc. New York, NY, USA.

[Callahan2005]
Callahan, E. (2005). Cultural similarities and differences in the design of university websites. Journal of Computer-Mediated Communication, 11(1) , article 12.

[Calvary2002]
Calvary, G., Coutaz, J., Bouillon, L., Florins, M., Limbourg, Q., Marruci, L., Paterno', F., Santoro, C., Souchon, N., Thevenin, D., Vanderdonckt, J. (2002) The CAMELEON Reference Framework, Deliverable 1.1 CAMELEON Project.

[Carroll2003]
John M. Carroll, Introduction: Toward a Multidisciplinary Science of Human-Computer Interaction, in HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science, Morgan Kaufmann, 2003. ISBN: 1–55860–808–7. 1--9.

[Collins1995]
Collins, D. 1995. Designing object-oriented user interfaces. The Benjamin/Cummings Publishing Company, Inc, Redwood City.

[Coutaz2012]

Coutaz, J., Calvary, G. (2012) hci and software engineering for user interface plasticity (2012). Chapter 52 in "The Human-Computer Handbook–Fundamentals, Evolving Technologies, and Emerging Applications", 3rd edition, Julie, A. Jacko ed., CRC Press Taylor and Francis Group, p.1195--1220.

[Curtis1994]

Curtis, B. and Hefley, B. 1994, January. A WIMP no more, the maturing of user interface engineering. Interactions 1, 1, 22–34.

D

[Deuff2012]

Deuff D. and Cosquer M., (2012) "Méthode agile centrée utilisateurs", Conférence ergoIHM 2012, Biarritz, October 17-19.

[Diaper2004]

Diaper, D. (2004) Understanding Task Analysis for Human-Computer Interaction. The Handbook of Task Analysis for Human-Computer Interaction, Ed. Dan Diaper, Neville A.Stanton, Lawrence Erlbaum Associates, pp. 5-47.

[Diaper2004b]

Diaper, D., Stanton, N. (2004) Wishing on a sTAr: The Future of Task Analysis. The Handbook of Task Analysis for Human-Computer Interaction, Ed. Dan Diaper, Neville A.Stanton, Lawrence Erlbaum Associates, pp. 603-619.

[Dijkstra1976]

E.W. Dijkstra, (1976) A Discipline of programming, Prentice Hall, Englewood Cliffs, NJ.

[Dittmar2000]

Dittmar, A. (2000). More precise descriptions of temporal relations within task models. In P. Palanque & F. Patern`o (Eds.), Proceedings of Seventh International Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS 2000) (Lecture Notes in Computer Science, Vol. 1946; pp. 151–168). Berlin: Springer-Verlag.

F

[Farenc1997]

Ch. Farenc, ERGOVAL : une méthode de structuration des regles ergonomiques permettant l'évaluation automatique d'interfaces graphiques, thèse de doctorat de 3ième cycle, Université de Toulouse I, January 1997.

G

[Gharsellaoui2014]

Gharsellaoui, A., Bellik, Y., Jacquet, C. Un système d'aide et de suivi des tâches utilisateur dans un environnement ambiant. IHM 2014, Oct 2014, Lille, France. pp.130-138, Actes de la 26e conférence francophone sur l'Interaction Homme-Machine. <hal-01080244>

[Grislin1997]

Grislin, M., Kolski, C. (1997) Proposition d'une démarche d'évaluation a priori des IHM de supervision. Journal Européen des Systèmes Automatisés (RAIRO-APII-JESA), 31 (7), pp. 1111-1154, 1997.

[Green1985]

Green, M. The University of Alberta User Interface Management System. Proc. Of the 12th Annual Conference on Computer Graphics and Interactive Techniques, pp. 205-213, 1985.

[Guerrero2008]

Guerrero, J., Vanderdonckt, J., Gonzalez Calleros, J.M. Towards a Multi-User Interaction Meta-Model. Working paper 08/28 UCL, Louvain School of Management. December 2008.

I

[IBM1993]

Object-Oriented Interface Design: IBM Common User Access Guidelines, document SC34-4399, International Business Machines, Publisher Que Corp., 1993.

[ISO2010]
ISO/IEC, "ISO/IEC 25010 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models," International Organization for Standardization, Tech. Rep., 2010.

[ISO2010]
ISO/WD 9241 Ergonomic requirements for Office Work with Visual Displays Units, International Standard Organization, 1992.

[ISO9241]
ISO FDIS 9241-210 Human-centred design process for interactive systems. ISO.2009.

H

[Hall1989]
Hall, E. T., Beyond culture. New York: Doubleday. 1989.

[Henningsson2002]
K. Henningsson and C. Wohlin, "Understanding the Relations between Software Quality Attributes - A Survey Approach", Proceedings 12th International Conference for Software Quality, Ottawa, Canada, Proceedings on CD, October 2002.

[Hofstede1997]
Hofstede, G., Culture and organisation: Software of the Mind. New York: Mc-Graw Hill. 1997

[Hutch2011]
Andrew Hunt, David Thomas, The Pragmatic Programmer, The: From Journeyman to Master. Addison Wesley, First Edition October 13, 1999. ISBN: 0-201-61622-X, pp.37

[Hutch2011]
Hutchinson, J., Whittle, J., Rouncefield, M., and Kristoffersen, S. (2011) Empirical assessment of MDE in industry, Proceedings of the 33rd International Conference on Software Engineering ICSE'2011, ACM Press, New York, pp. 471-480.

J

[Jacobson1999]

I. Jacobson, G. Booch and J. Rumbaugh. (1999) The Unified Software Development Process. Addison-Wesley.

[Johnson1989]

Johnson, P., & Johnson, H. (1989). Knowledge analysis of task: Task analysis and specification for human-computer systems. In A. Downton, (Ed.), Engineering the human-computer interface (pp. 119–144). London: McGraw-Hill.

K

[Kieras2004]

Kieras, D. GOMS Models for Task Analysis. The Handbook of Task Analysis for Human-Computer Interaction, Ed. Dan Diaper, Neville A.Stanton, Lawrence Erlbaum Associates, pp. 83-116. 2004.

[Khaddam2014]

Towards a Culture-Adaptable User-Interface Architecture. Revista Romana de Interactiune Om – Calculator. Vol.7, Num 2. Special Issue: User interface standardization 2014. ISSN 1843-4460 (pp. 161 – 194).

[Khaddam2015a]

Khaddam, I., Mezhoudi, N., Vanderdonckt, J. A Linguistic Perspective to Develop Graphical User Interfaces. 3ed International Conference on Control, Engineering & Information Technology (CEIT). Tlemcen, Algeria, 25-27 May, 2015.

[Khaddam2015b]

Khaddam, I., Mezhoudi, N., Vanderdonckt , J. Towards a Linguistic Modeling of Graphical User Interfaces: Eliciting Modeling Requirements. 3ed International Conference on Control, Engineering & Information Technology (CEIT). Tlemcen, Algeria, 25-27 May, 2015.

[Khaddam2015c]

Khaddam, I., Mezhoudi, N., Vanderdonckt , J. (2015) Towards Task-Based Linguistic Modeling for designing GUIs. 27ème conference francophone sur l'Interaction Homme-Machine. Oct 2015,

Toulouse, France. ACM, IHM-2015, article 17. ISBN: 978-1-4503-3844-8.

[Khaddam2016]
Khaddam, I., Barakat, H., Vanderdonckt , J. Enactment of User Interface Development Methods in Software Life Cycles. ROCHI'16, September 8–9, 2016, Iasi, Romania ISBN 978-1-4503-4322-0. DOI: http://dx.doi.org/.

[Kleppe2003]
Kleppe, A. G., J. Warmer, et al. 2003 MDA Explained: The Model Driven Architecture: Practice and Promise, AddisonWesley Longman Publishing Co., Inc.

[Klug2005]
Klug, T., Kangasharju, J. Executable Task Models. TAMODIA'05, September 26–27, 2005, Gdansk, Poland. Copyright 2005 ACM 1-59593-220-8/00.

[Kolski1998]
Kolski, C. 1998, May. A "call for answers" around the proposition of an HCI-enriched model. Software Engineering Notes 3, 23, 93–96.

[Kuhn1996]
Kuhn, Thomas S. (1996). The Structure of Scientific Revolutions (3rd ed.). University of Chicago Press. ISBN 0-226-45807-5. LCCN 96013195.

L

[Limbourg2004]
Limbourg, Q., Vanderdonckt, J. Comparing Task Models for User Interface Design, The Handbook of Task Analysis for Human-Computer Interaction, Lawrence Erlbaum Associates, Mahwah, 2004. pp. 135-154.

[Limbourg2009]
Limbourg, Q., Vanderdonckt, J. Multi-Path Transformational Development of User Interfaces with Graph Transformations, in Seffah, A., Vanderdonckt, J., Desmarais, M. (eds.), "Human-Centered Software Engineering", Chapter 6, HCI Series, Springer, London, 2009, pp. 109-140.

M

[Mahfoudhi2001]
Mahfoudhi, A., Abed, M.,&Tabary, D. (2001). From the formal specifications of user tasks to the automatic generation of the HCI specifications. In A. Blandford, J. Vanderdonckt, & P. Gray, (Eds.), People and computers XV (pp. 331–347). London: Springer.

[Mariage2005a]
Mariage, C.,Vanderdonckt, J., Chevalier, A. Using the MetroWeb Tool to Improve Usability Quality of Web Sites. LA-WEB 2005. pp. 213-222. 2005.

[Mariage2005b]
Mariage, C.,Vanderdonckt, J., Pribeanu, C. State of the Art of Web Usability Guidelines. The Handbook of Human Factors in Web Design. Lawrence Erlbaum Associates (Mahwah). pp. 688-700. 2005.

[Masson2010]
Masson, D., Demeure, A., Calvary, G. Magellan, an evolutionary system to foster user interface design creativity. In Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems (EICS '10). ACM, New York, NY, USA, 87-92.

[Mayhew1992]
D.J. Mayhew, Principles and Guidelines in Software User Interface Design, Prentice-Hall, Englewood Cliffs, 1992.

[Meixner2011]
Meixner, G., Paterno, F. & Vanderdonckt, J. (2011), Past, Present, and Future of Model-Based User Interface Development., i-com 10 (3) , 2-11.

[Motti2013]
Genaro Motti, V., Raggett, D., Van Cauwelaert, S., Vanderdonckt, J. (2013), Simplifying the development of cross-platform web user interfaces by collaborative model-based design. SIGDOC '13 Proceedings of the 31st ACM international conference on Design of communication. New York, pp.55-64. ISBN: 978-1-4503-2131-0.

[Lim1994]
Lim, K. Y., & Long, J. (1994). The MUSE method for usability engineering. Cambridge: Cambridge University Press.

[Myers1995]
Myers, B. User Interface Software Tools. ACM Transactions on Computer-Human Interaction, 2(1), pp.64-103,1995.

O

[Okes2009]
Okes, D. Root Cause Analysis. The Core of Problem Solving and Corrective Action. Quality Press, Milwaukee 53203. pp.4-7. ISBN 978-0-87389-764-8. 2009.

[Oliveira2015]
Oliveira, K., Girard, P., Gonçalves, T., Lepreux, S., Kolski, C. (2015) Teaching Task Analysis for User Interface Design: Lessons Learned from Three Pilot Studies. 27ieme conference francophone sur l'Interaction Homme-Machine., Oct2015, Toulouse, France. ACM, IHM-2015, article 31. ISBN: 978-1-4503-3844-8.

[Nielsen1986]
Nielsen, J. A. (1986) Virtual Protocol Model for Computer-Human Interaction. International Journal of Man-Machine Studies, March 1986, Vol.24, Issue.3 , pp. 301-312.

P

[Parnas1972]
D.L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM, Volume 15, No. 12, 1972

[Paterno1997]
Paterno, F., Mancini, C., Meniconi, S.: ConcurTaskTree: a Diagrammatic Notation for Specifying Task Models. In: Proceedings of IFIP TC 13 Int. Conf. on Human-Computer Interaction (Syndey, June 1997). Chapman & Hall, London (1997), 362–369

References

[Paterno2000]
Paterno, F., Model-Based Design and Evaluation of Interactive Applications.. Springer-Verlag London 2000. ISBN 978-1-85233-155-9

[Pilemalm2012]
Pilemalm, S., Hallberg, N., Sparf, M., and Niclason, T. (Dec. 2012) Practical experiences of model-based development: Case studies from the Swedish Armed Forces, Systems Engineering , Volume 15 Issue 4, pp. 407-421.

[Pisano1993]
Pisano,A., Shirota, Y. & Iizawa, A. (1993) Automatic generation of graphical user interfaces for interactive database applications. Proceedings of CIKM '93. ACM Press. P.344-355.

[Pribeanu2006]
Pribeanu, C. (2006) Task Modeling for User Interface Design - A Layered Approach. International Journal of Information Technology, 3(2), pp.86-90.

R

[Rau2011]
Rau, P.-L. P., Plocher, T., & Chong, Y.-Y. (2011). Cross-Cultural Web Design. In Handbook of Human Factors in Web Design. Second Edition (pp. Ch.33. pp.677-698). London: Tylor & Francics Group, CRC Press.

[Reinecke2010]
Reinecke, K. B. (2010). Modeling a User's Culture. In E. Blanchard, & D. Allard (Eds.), Handbook of Research on Culturally-Aware Information Technology: Perspectives and Models , pp. pp. 242-264.

[Royce1970]
W. Royce. Managing the development of large software systems. In proceedings of the IEEE WESCON, pp. 1-9, august, 1970

S

[Scapin1989]

D.L. Scapin, Guidelines for User Interface Design : Knowledge Collection and Organization, Technical Report ITHACA.INRIA.89.D12.03, Institut National de Recherche en Informatique et en Automatique, Rocquencourt, December 30, 1989.

[Schwaber2001]

Schwaber K. and Beedle M. (2001) Agile Software Development with Scrum. Prentice Hall, Upper Saddle River, NJ.

[Stanton2006]

Stanton, N.A., (2006) Hierarchical task analysis: Developments, applications, and extensions. Applied Ergonomics 37. Elsevier. pp. 55–79.

[Streveler1990]

Streveler, D.J., Wasserman, A.I. (1990) Quantitative Measures of the Spatial Properties of Screen Designs, in: Proceedings of 3rd IFIP TC13 Conference on Human-Computer Interaction (INTERACT'90), Cambridge, August 27-31, 1990, D. Diaper, D. Gilmore, G. Cockton, B. Shackel (Eds.), Elsevier Science Publishers, Amsterdam, 1125–1133.

T

[Tran2012]

Tran, V., Vanderdonckt; J., Tesoriero R., Beuvens, F. (2012). An Analytical Benchmarking of Algorithms for Generating Abstract User Interfaces, EICS'12. Copenhagen, Denmark, June 25–28, 2012. pp. 101-110.

[Tylor2010]

Richard N.Taylor, Nenad Medvidović, Eric M. Dashofy (2010). Software Architecture - Foundations, Theory, And Practice. John Wiley & Sons, Inc. 2010. ISBN-13 978-0470-16774-8.

U

[UsiXML2007]

UsiXML Consortium. UsiXML, a General Purpose XML Compliant User Interface Description Language, UsiXML V1.8, 23 February 2007.

Available on line: http://www.usixml.org/

V

[VanDerVeer1999]

van der Veer, G. C., Van der Lenting, B. F., & Bergevoet, B. A. J. (1996). GTA: Groupware task analysis-modeling completity, Acta Psychological, 91: 297–322.

[Vanderdonckt1999]

Vanderdonckt, J. (1999). Development Milestones towards a Tool for Working with Guidelines. international Journal of Man-Machine Studies, 24 (3), 301-312.

[Vanderdonckt2007]

Vanderdonckt, J. (2007). Guide ergonomique des interfaces homme-machine. Namur, Belgique: Presses Universitaires de Namur. ISBN : 2-87037-189-6

[Vanderdonckt2008]

Vanderdonckt, J. Model-Driven Engineering of User Interfaces: Promises, Successes, Failures, and Challenges. ROCHI'08. Iasi, Romania. Sept. 18-19, 2008.

[VanWelie1998]

van Welie, M., van der Veer, G.C., Eliëns, A. An Ontology for Task World Models. In Proc. of 5th International Eurographics Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS98 (3-5 June 1998, Abingdon, UK), pp. 57-70.

W

[Weyns2004]

D. Weyns, A. Helleboogh, E.Steegmans, T. De Wolf, K. Mertens, B.Boucké and T. Holvoet. Agents are not part of the problem, agents can solve the problem. In Proceedings of the OOPSLA Workshop on Agent-Oriented Methodologies, 2004

References

Appendix A. Guerrero's Task Identification Criteria

Guerrero *et al* [Guerrero2008] established identification criteria for tasks in the context of supporting developing user interfaces for workflow. The workflow model describes how the work in the organization flows by defining models of: process (what to do), tasks (how to do it), and the organizational structure (where and who will perform it). A workflow model has at least one process and a process has at least one task. The process model indicates the ordering of tasks in time, space and resource. The task model is employed to support developing the user interface.

Guerrero identification criteria help to identify tasks in the process model. Tasks in the process model represent the leaf nodes in the process hierarchy. These criteria are:

1- Change of space (or change of location): when the scenario indicates a change of location of the operations, a change of task may occur. Therefore, any scenario fragment like "in the headquarters, the worker does …, then in the local agency, the worker does…" indicated a change of space, therefore a change of task. The main location where the task is carried out takes the advantage.

   In case of collaborative or cooperative tasks, the main location is considered to detect whether there is any change of location.

2- Change of resource: when the scenario suggests that new or different resources are exploited, a change of task may occur. We distinguish three categories of resources:

   a. Change of resource of type "User stereotype": when another user stereotype appears in the scenario may indicate that there is a change of task. For example, "a clerk does …, then an employee files the results of …". The two

different names for two different users indicate a change of task. This reasoning always forces identifying who is the main responsible person for carrying out a task. Again, in case of collaborate or cooperative tasks, the main user stereotype involved in the task is selected.

b. Change of resource of type "material": when another material resource appears in the scenario, a change of task may occur. For example, "a clerk enters the customer's data on a PocketPC, and then takes a picture with a mobile phone camera" indicates two tasks resulting from the usage of two different resources, here a PocketPC and a mobile phone. This should not be confused with a task that is performed on different computing platforms, like in the context of a multi-device UIs. Thus, any significant change of software and/or hardware resource may indicate that there is a change of task.

c. Change of resource of type "immaterial": when another immaterial resource appears in the scenario, a change of task may occur. For example, "a network administrator uses a specific software to check network status; s/he uses another software to update the computers of the network". The two different types of software involved indicate a change of task.

3- Change of time: when the scenario indicates a different time period in which the task is performed, a change of task may occur. We differentiate four criteria resulting from any potential change of time:

a. Existence of an interruption: when the task is interrupted by an event that changes the time period. For instance, "an employee registers every incoming complaint. After registration a form is sent to the customer who returns the form within two weeks". A task can be interrupted

for many reasons, such as an error, an external event, a dynamic task.

b. Existence of a waiting point: when in the development of a task there is a moment where it is necessary to wait that something occurs for continuing, a change of task may occur. We have two types of waiting points:
    i. Waiting point of type "decision": when a determination arrives at after consideration of a question, a change of task may occur. For example, "after the preparation of a flight plan, the pilot will take the decision to fly". A decision can be made by a human agent, a system agent or in a mixed-initiative way. In any case, there could be as many different tasks as there are different alternatives coming out the decision. In this way a decision could result into two tasks after the decision or multiple tasks (like in a "case of").
    ii. Waiting point of type "accumulation": when there is necessary to create a waiting list for some information, a change of task may occur. For instance, "due to a car accident, more complaints arrived yesterday at the insurance agency and the employee had to register all incoming complaints to send as a group to directors". The accumulation may be related to documents (or messages, or data) or to processes (e.g., a repetition of tasks).
c. Permanence of execution unit: when the task execution depends of the results of at least two previous asynchronous tasks. For instance, "the results of an insurance complaint are delivered to the client when the complaint manager provides whether the complaint applies or not and when the evaluator provides the estimated cost".

d. Periodicity of execution: when there are different periodicities established to execute tasks, then a change of task may occur. For example, "every Monday the employee does a backup of the information". This criteria is often detected when one can determine that the frequency of one task is different from the frequency of another. For instance, a backup (automatic) task could be incremental every day and full each Friday. In this case, we separate two tasks (incremental vs total backup) because their frequencies are different: every day vs. every Friday.

4- Change of nature: when the scenario represents a change of category, a change of task may occur. A task may have any of the following nature: manual, automated, interactive or mechanical. Any change of this nature may indicate a change of task. For instance, "first a secretary types a letter in the computer (interactive), after a printer prints the text (automatic) and finally the manager signs the letter (manual)".