Using EUREQA for End-User UML Model Development through Design Patterns

Paul G. Austrem
Dept. of Information Science and Media Studies, University of Bergen, Bergen, Norway
Email: paul.austrem@infomedia.uib.no

Abstract—**This work presents the EUREQA approach for end-user development. The purpose of the tool is to narrow the cognitive gap between the end-user developer's mental model and the software model. The tool uses design patterns as building blocks allowing end-users to create UML class diagram models that capture their domain knowledge. The EUREQA tool hides from view the complexity of code thereby reducing the cognitive load on end-user developers. EUREQA makes non-functional quality issues a first-class concern allowing end-user developers to consider both functional and non-functional aspects of design alternatives. The tool uses visualization techniques to aid in non-functional quality assesment. The purpose of this paper is to evaluate and assess the opportunities and challenges of EUREQA. A qualitative, pilot evaluation of EUREQA shows that the visualization techniques work well, whereas there are issues with the abstraction gap between the visualization and class diagram.**

Index Terms—End-User Development, Tool-support, Design Patterns, Non-Functional Qualities

I. Introduction

End-user development (EUD) is defined as the activity of an end-user of a software system being partially or conpletely in involved in the development effort. Moreover, end-user development is a growing domain. Numbers from [1] estimate that there in 2012 will be more than 55 million end-users in American workplaces, with 13 million of these performing programming activities. This is in contrast to the expected 3 million professional software developers in the US. It is evident that end-user development is an important and ubiquitous domain.

In spite of these numbers most software development tools are designed with software professionals in mind. However, the fundamental design of professional software development tools is incongruent with the needs of an end-user developer. First of all, professional development tools have a high skill threshold and assume a high degree of knowledge of its users in order to be used productively. Secondly, professional development tools requires the user to think in terms of the computer model and some paradigm such as object-orientation. These are concepts that are alien to an end-user developer, and give rise to an issue identified as the cognitive gap defined as the distance between the end-user developer's mental model and the computer model [2]. A similar concept is the communications gap, defined as the difference in mental models between end-users and software professionals [3].

We argue that among the expected 13 million end-user programmers, many will be working with mobile technology. An increasing number of businesses are using mobile technology to improve work processes for professionals who perform their work tasks away from a desk with a desktop computer. This can be observed in public transport with for instance trains where ticket conductors print tickets and find schedules on a mobile device. In the medical domain, doctors can use mobile devices to retrieve real-time patient journal data [4], and mobile computing can be highly effective in logistics [5] and aircraft maintenance [6]. However the introduction of mobile work also introduces new challenge as new requirements may only emerge after deployment and use "in the wild" [7]. Thus the introduction of moble technology will further increase the need for end-user development tools, techniques and methods.

Developing software solutions that support mobile work requires domain knowledge and work process knowledge. The professionals using mobile solutions are the people best equipped to contribute in the area. Traditional approaches may involve consultancy companies or third parties performing software development beginning with requirements elicitation through to design and on to coding, testing and deployment. Even with agile methods it is arguably a time consuming effort involving also management planning and approval processes that create further overhead. Principally end-user development inverses this approach. Instead of software consultants learning about the domain from users, end-user developers become familiar with software design. End-user development is defined as "a set of methods, techniques and tools" [8] (p.2) which allow users of software systems to "at some point create, modify or extend a software artifact" [8] (p.2).

In a mobile setting the domain knowledge becomes even more important, because the context in which the work is conducted will more strongly affect requirements. For example, a home healthcare worker filling in data on a stationary computer will have no environmental distractions, whereas someone performing the same task

doi:10.4304/jsw.6.4.690-704

on-site during a visit will simultaneously have to be aware of and tend to the needs of the patient. Thus a different context arises than if the person was working at a stationary computer. EUREQA is not designed specifically to deal with end-user development of mobile systems, but is motivated in part by the growing need through mobile technology to involve end-users in the development process.

This work is part of the M3W project focusing on model-driven support for multi-channel mobile work. EUREQA is part of the model-driven area of the project and uses design patterns as foundational building blocks from which UML class diagram models can be constructed. In this work, we present EUREQA with results from a qualitative pilot evaluation.

II. Theoretical Foundations of EUREQA

The benefits of end-user development are multi-faceted. Firstly, the end-users get a sense of system ownership. Reporting on first hand experience Wagner and Picolli show that it is imperative that developers listen to end-users, accommodate participatory design and that user participation can be most valuable and powerful after they have started using the system [9].

Wagner and Picolli [9] along with Klaus, Wingreen and Blanton [10] report that end-users can topple multimillion dollar projects through adoption resistance. The root of adoption resistance has been investigated for decades. A survey by Hirschheim and Newman indicated that resistance is inherently complex, but a few aspects in their work seem equally relevant today as they did twenty years ago [11]. Firstly, they identified lack of involvement in the change and lack of felt need as concerns. These are issues that pervade software development to this day. Similarly they state that organizational invalidity is a contributor to user resistance. Organizational invalidity equates to process re-engineering in order to accommodate the changes induced by introducing information technology. If the changes made to work processes feel in some way awkward for the user then this could raise tensions and spur adoption resistance.

The second salient benefit is a reduced cost of development. Development efforts that are reliant on external suppliers and consultants who do not have any tie-in to the day-to-day operations will produce administrative and managerial overhead. Thus, any post-deployment development effort run by an external supplier will increase the time between a requirement is identified from field use until it is functionally resolved. Moreover, there may be cost issues related to the exclusive use of external suppliers and consultants compared to relying on in-house end-user development efforts.

Given the aforementioned benefits of end-user driven development, there are a few challenges in end-user development that EUREQA attempts to resolve. Fischer, Giaccardi, Ye, Sutcliffe and Mehandjiev report that encouraging end-user development is from a managerial point of view risky or even outright hazardous as it could lead to the introduction of unreliable and unmanageable software [12]. Costabile et al. proposed that “there is a high level of errors in applications developed by end-users” [3] (p. 6). Segal tenders that a major problem in end-user development effort is the narrow focus on achieving the functional goal(s) and ignoring everything else, such as non-functional qualities (NFQ) [13]. This claim is supported by Chung and Leite who describe it as a *lop-sided* emphasis on functional requirements [14]. When even professional developers tend to ignore non-functional requirements there is little hope that end-user developers will pay any attention to them either as they are primarily occupied with creating a solution that satisfies their functional needs. However there are strong reasons for focusing on non-functional requirements, with numerous examples of the significant costs of ignoring them. Projects such as the London Ambulance System, the Mars Climate Orbiter and a licensing system for the New Jersey Department of Motor Vehicles were all scrapped or deemed failures due to not meeting non-functional requirements according to Kassab [15].

The non-functional requirement failures described above may not all be equally pertinent to end-user development as some might have failed due to performance-related non-functional requirements. Mehandjiev, Sutcliffe, and Lee report that security, data accuracy, maintainability and reliability of software developed by end-user developers are all major concerns from an ogranizational point of view [16].

Mørch, et al. state that from an EUD perspective, professional development tools and IDEs (for example Visual Studio or Eclipse) lack certain qualities that must be present in an EUD tool [17]. They state that an EUD tool should a) offer “metaphors that provide meaningful abstractions for end-users, allowing them to break up applications into suitable components and assemblies” [17], (p. 60) and b) offer the possibility for end-user developers to gradually learn to build or modify software components. This is referred to as the *gentle slope theory*, a design heuristic which stipulates that moving from one level of abstraction or activity to the next should not incur a sharp increase in the environment's complexity or in the skills required of the user [8]. End-user development is an increasingly important domain and currently most software development tools are designed with professional developers in mind. A major challenge is providing an environment that does not cognitively overload the end-user developer. This can be achieved through limiting a tool to only dealing with a portion of the software development life-cycle. End-user development environments should provide a gentle slope. Another design goal of end-user development tools is reducing the cognitive gap, that is providing a development environment which aligns with the end-user developer's mental model. A final concern is making end-user developers aware of the non-functional aspect of software artifacts.

The domain of end-user development is rich with various techniques, methods and approaches. A discussion and summary of the various techniques is

offered in [8]. A common distinction in end-user development is made between tools for use during *design-time* and *use-time*. Design-time tools encourage the end-user to partake or perform development before the software solution is implemented. Use-time tools are more often aimed at what is known as end-user *tailoring*, where an end-user adopts an existing artifact during use. Another dimension along which end-user development tools are separated is their level of abstraction. At the lowest level of abstraction are techniques such as scripting in spreadsheets, the most widespread technique currently in use. The benefits are that it is quick with little overhead and allows for immediate execution. The obvious drawbacks are a) that it requires the end-user developer to have a basic understanding of programming concepts such as variables and variable scope, along with learning script syntax and b) scripting does not scale well to larger solutions.

At the intermediate level of abstraction, one finds model-based and component-based approaches. Mørch et al. have done work on the components-based approach within end-user tailoring [17]. The principle is that an end-user developer can combine various components to create a software solution. The components act as containers of a pre-defined functional behavior. They can be combined through connections according to a set of rules. Won, Stiemerling and Wulf provide an implementation where they dub these connections ports with rule-based checking of connections [18]. A major benefit of the components-based approach is that it hides from view the inner workings and software complexity of each component. The end-user works at a higher level of abstraction relating only to descriptions of what the component does at a high functional level. Moreover the rule-enforced connections between components stops the end-user from making any syntactical errors. However a limitation of the component-based approach is the black box nature of the components and that end-user developers cannot natively express their domain knowledge in the environment. Another issue identified by Mørch et al. relates to how end-user developers deal with situations where the application framework does not provide the component they need [17]. This would require them to develop the component themselves.

At the highest level of abstraction are tools such as Hands [19]. High abstraction level environments are often domain-oriented and use modeling notation that are direct representations of the specific domain. This removes any issues caused by *closeness of mapping* [20] and aligns with an end-user developer's mental model. However the most obvious issue with such environments are their narrow scope and lack of flexibility. Moreover, if the tool is aimed for high-level model-based application development, it would further require a potentially complex, proprietary model tranformation engine for forward-engineering.

The three levels of abstraction described above are only a sample, with other approaches existing inbetween these. Within each level of abstraction are a plethora of techniques that enable end-user development. We will not go into details here as [8] provide a good descriptive coverage but some of the key techniques are visual languages, model-based development, programming-by-example, parameterization and annotation.

All of the three approaches above have their relative strengths and weaknesses, however a common denominator among all of them is that none of them deal explicitly with non-functional requirements. Again the non-functional aspect is treated as an inevitable consequence which one can do nothing about. Exceptions to this are few and far between. One example is the State of Oregon who uses an information system named Oregon Public Employees Retirement System (PERS). The system is used to manage pension payouts to former state employees. In their IS development guidelines, they provide a separate end-user standard development guidelines documentation that highlights non-functional requirements as a separate section [21].

Design patterns are an industry-wide approach to software design reuse. They also exist at an intermediate level of abstraction. Design pattern solutions are represented in UML class diagram form that requires little to no model transformation in order to be forward-engineered. At the same time they are presented in a higher level form using metaphors and analogies to real-world situations making them more understandable and focus on overall solution properties rather than specific technical details. Surprisingly design patterns have, to the best of our knowledge, not previously been used in an end-user development setting.

We propose the EUREQA tool for end-user development which aims to resolve some of the issues with the aforementioned tools and make non-functional requirements a first-class concern in end-user development. EUREQA uses design patterns as foundational building blocks from which an end-user developer can construct a UML model solution. The following section will outline the theoretical framework upon which EUREQA is built.

A. Design Patterns

Design patterns were introduced to the domain of software in 1994 through the seminal work of Gamma, Helm, Johnson and Vlissides [22] A design pattern is a solution to a problem within a certain context. We consider design patterns to be a strong candidate to resolve the issues presented earlier for three reasons. First of all, a key feature of design patterns is the use of *metaphors and analogies* in describing the technical solutions. They are not exact blueprints of implemented code but rather abstracted models based on object-oriented software principles. As such, they exist at a higher level of abstraction than UML models. Design patterns are often presented in the form of UML class diagrams wherein each class has a specific role within the overall behavior of the design pattern. The classes can be renamed to closer align with the domain the design pattern is being used in. Finally, the solution portion of a design pattern is often tried and tested, as such design patterns contain a certain level of quality assurance. A

notable trait of design patterns is that they expose the non-functional result of applying them. These are described through textual descriptions titled consequences. Design patterns therefore natively elucidate non-functional requirements and bring them to the forefront of consideration in design decisions. In EUREQA, we leverage design patterns because of these traits and use them as building blocks for UML class diagram model construction.

B.Non-functional requirements

As discussed, non-functional requirements are important in any non-trivial software solution, and if neglected can have disastrous consequences. They are also a challenge on a conceptual level. There is no mutual agreement on what they are although many definitions have been proposed. Although standard definitions have been proposed by bodies such as ISO [23] but there is still contention in the research community as to whether they are accurate and complete. Chung and Leite deliberate the various definitions and tender that the most precise one is "NFRs constitute the justifications of design decisions and constrain the way in which the required functionality may be realized" [14] (p. 366).

One of the reasons non-functional requirements are neglected is because they are hard to measure and quantify. Therefore one often aims to satisfice non-functional requirements rather than objectively satisfy them through metrics. Glinz discusses the complexity of non-functional requirements and proposes a taxonomy separating between various types of non-functional requirements [24]. One group is what he dubs "quality attribute requirements". A non-functional quality attribute requirement, often known as "ilities", is "a requirement that pertains to a quality concern other than the quality of meeting the functional requirements" (p. 4). This is a somewhat ambiguous definition, but it only highlights the inherent ambiguity of quality attributes. They cannot be directly operationalized the same way performance requirements can. Performance requirements can be quantitatively measured through CPU cycles or data throughput. Quality attributes on the other hand can be qualitatively assessed. Glinz states that for qualitative verification "no direct verification [is possible]. May be done by subjective stakeholder judgment of deployed system, by prototypes or indirectly by goal refinement or derived metrics" [24] (p. 4). The non-functional requirements identified by Mehandjiev et al. as major issues in end-user development all belong to the category of non-functional quality attributes [16].

EUREQA aims to deal specifically with quality attribute non-functional requirements. First of all, this is because most of the projects mentioned earlier failed due to quality attribute neglection. This highlights the importance of dealing with them to sustain a software solution long-term. Secondly, and as the next section will show, previous work has investigated the use of design patterns as a way of dealing with these types of non-functional requirements.

C. Using design patterns to satisfice and trace non-functional requirements

Gross and Yu [25] explored the use of design patterns as a means of satisficing non-functional requirements. Specifically they proposed systematic treatment of non-functional requirements in design pattern descriptions and analyzing them using the NFR-Framework [26]. Cleland-Huang [27] proposes design patterns as a means of achieving non-functional traceability in software solutions. Hsueh and Shen [28] propose a "pattern-aided approach to handling non-functional requirements and assisting the resolution of conflicting requirements" (p. 614), whereas Cleland-Huang and Schmelzer [29] elaborated on their approach by using design patterns as trace artifacts between a soft goal interdependancy graph and the underlying object-oriented model. The approach was based on creating user-defined links between classes, or class clusters, and non-functional requirements. The class clusters are the group of classes belonging to a design pattern. The underlying theory is that "*if an NFR is implemented through a design pattern, and if that design pattern can be detected, then finely grained traceability links can be generated*" [29] (p. 6). Fletcher and Cleland-Huang [30] propose using design patterns within a soft-goal framework as design solution candidates. The design pattern is then "contextualized" according to the given constraints and context of the specific problem. As such the generated UML class diagram operationalizes the softgoals.

As we can see, some work has been done showing how design patterns can be used to deal with non-functional requirements. Simultaneously design patterns exhibit many traits which are desirable in an end-user development setting. However, no work has been done which uses design patterns and deals with non-functional requirements in an end-user development setting.

III. Method

This work employs design science [31][32] as its main research methodology. The reason for choosing design research as the method is because it is a widely applied research approach when dealing with novel and new technologies / techniques and allows the focus to be on the designed artifact. The development effort itself is part of the epistemological basis of design science research. The development of the tool has been driven by specifications and requirements emerging through extensive literary review.

March and Smith [33] propose four artifacts that can result from design research in information systems, namely constructs, models, methods, instantiations, wherein the EUREQA tool is an instantiation. The literature research that has been conducted provides the theoretical framwork on which the instantiation is built. Vaishnavi and Kuechler [34] present design science as an iterative research effort with each iteration involving five steps. The steps are *awareness of problem, suggestion, development, evaluation* and *conclusions*. Our awarenesss of problem, as discussed above, stems from

an identified lack of end-user development tools that are both oriented at model-driven development and consider non-functional requirements. From this we have formulated the research question *can the use of design patterns reduce the cognitive gap in end-user development?* The EUREQA tool and the theoretical foundations correspond to the *suggestion* step. The following section briefly presents the case problem used to evaluate EUREQA. This will be followed by a section describing the development of EUREQA, which corresponds to the *development* step described by Vaishnavi and Kuechler [34].

A separate section presents the evaluation that aligns with the *evaluation* step. We have used qualitative methods for the evaluation. The use of qualitative methods in design science evaluation is important. Hevner et al. [31] state that "the rich phenomena that emerge from the interaction of people, organizations, and technology may need to be qualitatively assessed to yield an understanding of the phenomena adequate for theory development or problem solving" (p .77). At the core of design science is the designed artifact. Using the qualitative technique of content analysis and coding in a design science research setting makes it natural to define constructs around the key properties or features of the designed artifact before data analysis begins. This is known as *a priori* construct definition and is described by Eisenhardt [35] and Flick [36]. A priori constructs can be generated through literature review and in our case from specific features of the designed artifact.

Finally the analysis and discussion sections correspond to the *conclusions* step of Vaishnavi and Kuechler's [34] model.

The case problem

In Bergen, Norway a large project is currently being undertaken to build the "Bybanen", a light rail system through the city center. In this work, we have conducted a semi-structured interview with an operations manager (OM) with one of the subcontractors involved in the building of the "Bybanen". The OM noted that a lot of time was spent and to a certain degree wasted as part of logistics and materials deliveries. Quite often materials would be addressed to a specific engineer on the project, however the engineer himself would not be there to sign for it, instead someone else would and the engineer waiting for the delivery would not be notified. The engineers carry with them mobile devices. Such challenges could be resolved with mobile technology. On all projects, the construction company utilizes a web solution called a "project hotel" where all progress is registered daily. The OM reflected that an ideal solution would be some sort of mechanism allowing an engineer to be notified whenever materials/resources arrived for him/her and was registered in the "project hotel" website. To summarize, the engineers require timely notifications whenever materials or resources arrive. These notifications should ideally be sent to their mobile devices.

IV. Development of EUREQA

During the development of the EUREQA tool requirements have emerged through internal testing and evaluation, thus these have spurred further literature review and development of new functionality creating a cyclic process, as described by Vaishnavi and Kuechler [34]. Functional requirements have also emerged through discussions and feedback from other researchers and colleagues.

Although design patterns explicate the non-functional consequences of their use and apply metaphors to make the underpinning software design principles more accessible, end-users may still find it challenging to assess the appropriateness of a given design pattern. This is because end-user developers will find it hard to decide on whether a given design pattern disqualifies itself due to its non-functional consequences. For inexperienced end-user developers, this is a major issue. They will not have the background knowledge required to select a design pattern given their non-functional requirements. To relieve this, a non-functional requirements-driven design pattern-based tool was developed to assist end-user developers in making coherent design choices reflecting the context and non-functional requirements of a system. The tool forces the end-user developer to consider the non-functional consequences of using a specific design, and provides advice and guidance until they have a complete design model from which code can be generated. The end-user developer creates a non-functional profile for the application reflecting the context of the domain in which the application will be used. This profile forms the basis upon which solutions in the form of design patterns are selected. The tool utilizes vsiualisation with radar charts and Goal-oriented Requirements Language (GRL) [37] models to show the non-functional consequences and appropriateness of various design patterns chosen from a design pattern repository.

The second approach is the use of EUD project history. When a user or a group of users work with the tool, they first have to define the project on which they are working. All design solutions they generate during their work session will be saved and appended to that specific project along with a log of all design decisions (as recorded in audio) they made. EUREQA encourages cross-project learning and knowledge distribution, since it also allows users to share design solutions and design decisions in the form of attached audio recordings.

EUREQA is based on a step-by-step approach wherein the user is required to perform certain tasks before progressing to the next step. This is to a) limit the amount of information shown in one screen and b) provide a gentle slope of increasing complexity by the user performing the easier tasks first before moving on to the more complex tasks. In EUREQA, this is implemented through a tab-based environment.

A. Step 1 in EUREQA – define a non-functional profile

The first tab the user works in is the *Main Tab*. In this tab the user performs one, or optionally two tasks. The

main task is defining their non-functional profile. The tool allows for the definition of a non-functional profile by quantifying each non-functional quality on an ordinal scale ranging from 0-10. This is done in the area with a white border in Fig. 1. The lowest value of 0 indicates that the non-functional quality is of no importance, and thus does not need to be supported by the design. Conversely, a value of 10 indicates that the non-functional quality is extremely important. An issue with this can be that users assign high values for all of the non-functional qualities, however this would lead to few, if any, design patterns satisficing their requirements. The set of non-functional qualities can not be extended by the end-user. However, the end-user can modify and reduce the set through the Setup tab.

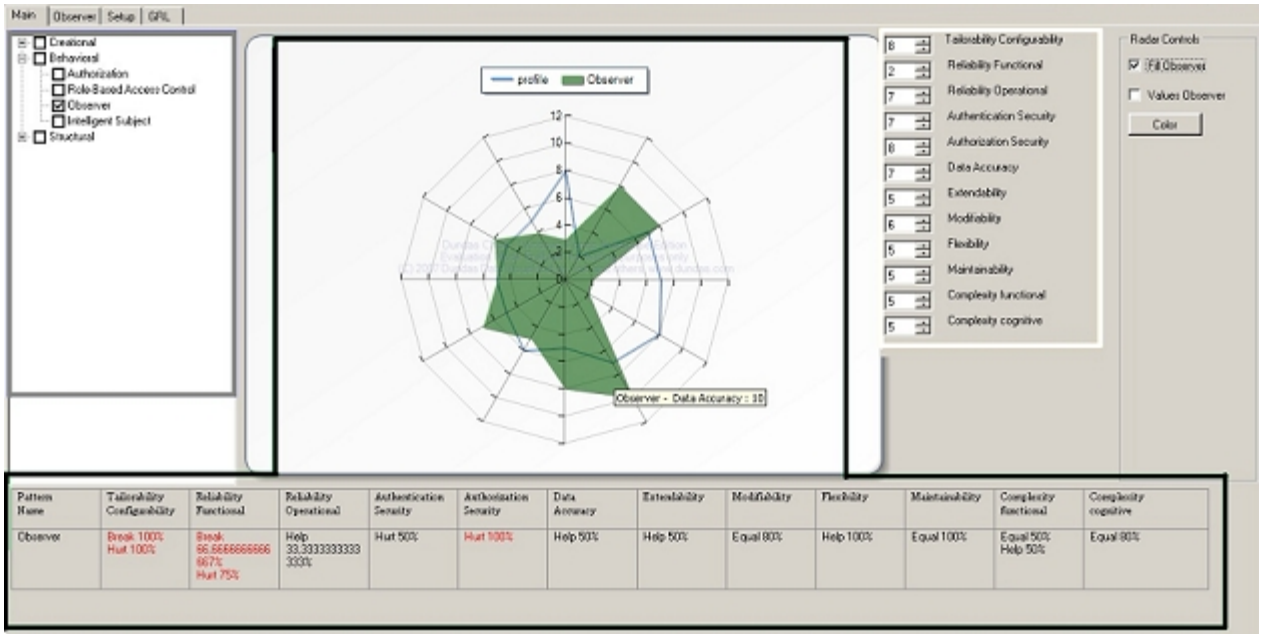

Figure 1. Main tab in the EUREQA tool.

The design patterns are retrieved from a repository (database) where their non-functional qualities also are quantified, indicating how well the given design pattern resolves each non-functional quality. The design pattern are listed in a library shown on the left-hand side of Fig. 1 with a grey border. The non-functional quality attribute values are presented in a radar chart where an overlay of the selected design pattern(s) show how well they match the user-defined non-functional profile. Below this is a table detailing the degree of support of each non-functional quality of the design pattern given the user-defined profile. The table uses the nomenclature of the GRL language [37] and allows for degrees of support through use of fuzzy sets. The radar chart and GRL fuzzy table are shown in the black border in Fig. 1.

B. Step 2 in EUREQA – Selecting a design pattern

The second step in the EUREQA approach is selecting a design pattern. In the pilot evaluation version of EUREQA we only allow for the selection of one pattern that can be manually augmented at a later stage on a class-by-class basis. Selecting a design pattern can be done in one of two places. The first is through the Main tab with the radar chart information as the foundation for making a decision. However, it is not guaranteed that end-user developers can manually do the mapping between the functional task that they wish to solve and the appropriate design pattern(s). Often several design patterns may resolve the same functional requirement(s) with the variation existing at the non-functional level. Martin [38] for instance shows the different approaches that can be used to solve the following problem "*...a need to add a new method to a hierarchy of classes, but the act of adding it will be painful or damaging to the design.*" [38] (p.525). This can be solved by various patterns, collectively known as the Visitor family of patterns [38]. But most end-user developers will not know that this functional problem statement can be solved using the aforementioned Visitor family of patterns.

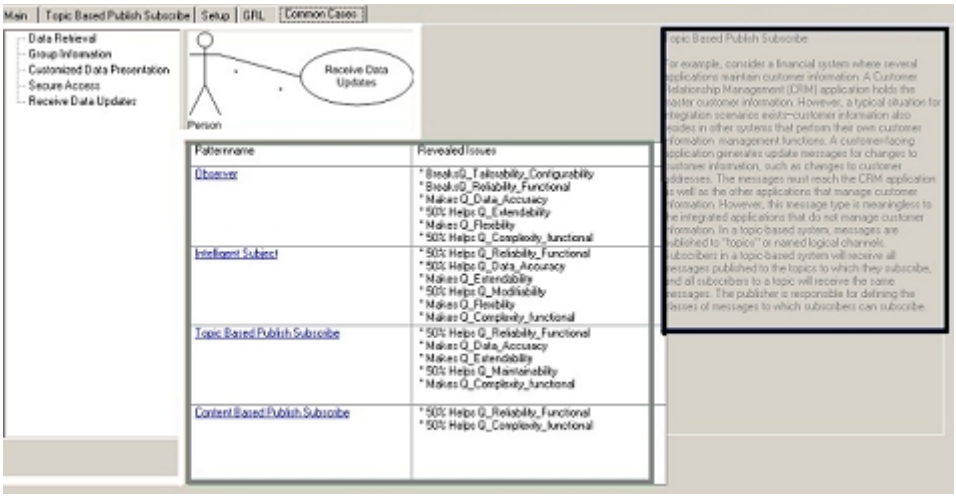

Figure 2. The Common Cases tab in EUREQA

Thus EUREQA provides a second approach to choosing a design pattern and way of mapping from the functional description to appropriate patterns in the *Common Cases* tab (see Fig. 2). Here, a selection of common functional tasks are shown in use case form. They are shown on the left in the white-bordered area of Fig. 2. When selecting one of the common cases, the tool will search the database for patterns that are related to the chosen common case and present a table containing the patterns. It will also match the profile of each design pattern against the current user-defined profile and indicate how the various patterns align with the profile. This is shown in the grey bordered area of Fig. 2. The user can also click on the name of a pattern and retrieve additional information such as the intent and purpose of the pattern, as shown in the black bordered area of Fig. 2. As such, the second step of the EUREQA approach is doable in two separate tabs.

C. Step 3 in EUREQA – Class diagram

The final step in EUREQA is working with the chosen design pattern at the class diagram level. This is done by selecting the tab with the name of the pattern the user has selected. Here the end-user developer can access the structural details of it through a Visio model (Fig. 3) and use domain specific naming to replace the standard naming of the design pattern classes.

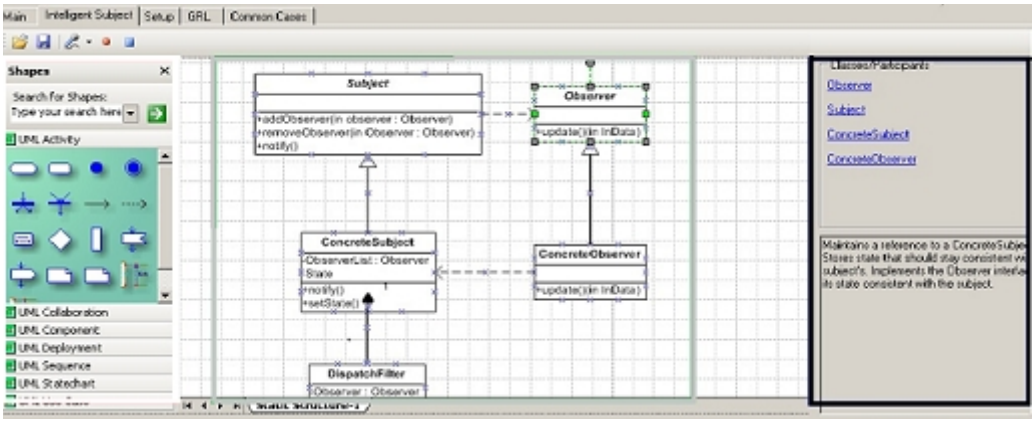

Figure 3. Class Diagram tab in EUREQA

The user can read information about the specific classes in the class diagram by clicking a class name contained in the black-bordered section on the right-side of Fig. 3. The tool offers support for audio recording of design rationale and decisions as they are made, for instance every time the end-user developer moves or changes an element in the model (s)he can record a short oral note on the why and how. The toolbar at the top of Fig. 3 shows the audio recording controls. This is all stored for future reference in a database linked to the specific session.

As mentioned, the tool currently does not provide for automated integration of other design patterns. In the pilot evaluation version, a UML toolbox is available providing the user with model elements to further work on the diagram by manually adding UML elements. The section on the left-hand side with a white border in Fig. 3 shows the UML toolbox. The grey bordered section in the middle is the class diagram canvas. Obviously there are inherent problems with manual UML manipulation. First of all, it is unlikely that end-user developers will have the required skillset to successfully manually add UML diagram elements with incurring syntactic and semantic errors. Secondly, manually adding UML elements, even if they are syntactically and semantically correct, can cause model inconsistencies in relation to the overall

solution satisficing non-functional requirements. However, for our protoype evaluation, we are mainly interested in the novel functional aspects of the tool such as defining a non-functional profile, the radar chart, Common Cases and their use of design patterns. Offering the users the possibility to manually add to the design patterns in the class diagram can give us a gauge of their confidence in the design pattern model solution.

D. GRL Tab and Model

EUREQA provides a GRL view model in a separate tab using a subset of the GRL ontology [37]. It provides a different perspective on the non-functional effects of the selected design pattern by using elements from the GRL language such as *Task, Softgoal* and *Contribution relationships* to indicate how the pattern affects the various non-functional characteristics relative to the user-defined profile.

The use of GRL is motivated by its semantic simplicity furthered by the use of a subset of its ontology. The GRL model allows for a larger perspective, providing a holistic view of how the pattern affects the various non-functional requirements. End-users can utilize the GRL map of the design pattern they have selected to see if any non-functional requirement is becoming overly burdened.

E. Fuzzy set support in the tool

When working with software design, every design decision is a trade-off. Rarely do you encounter cases where it is either completely apparent to either use or discard a solution. Most of the time there are some benefits and some drawbacks to using a design solution, often in the form of non-functional conflicts. Moreover a design solution may provide *some* support for a given non-functional requirement, but one is dealing with degrees of support rather than an either/or scenario. To remedy this, the tool applies fuzzy set theory to aid with the inherent vagueness and ambiguity of non-functional quality attributes that may confuse the novice. Fig. 4 shows the five fuzzy sets for the *Reliability Operational* non-functional attribute of the *Observer* pattern.

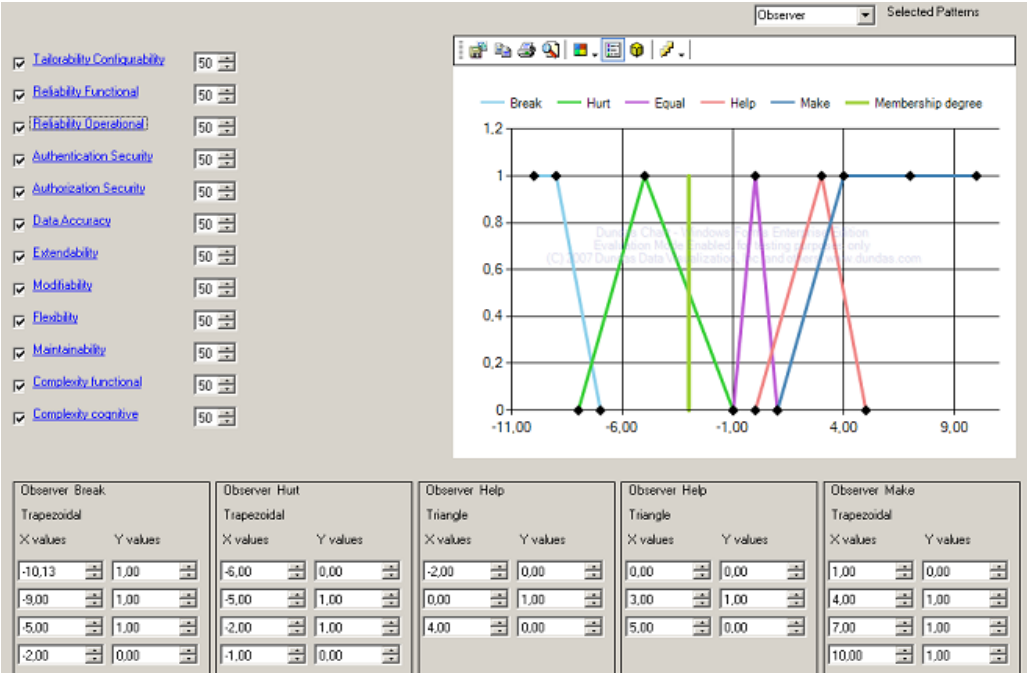

Figure 4. Fuzzy set configuration tab

In Fig. 4 a screenshot of the fuzzy set setup is shown. The EUREQA tool allows for complete customization of the fuzzy sets which constitute the basis for recommendations on specific non-functional characteristics of individual patterns in case the end-user developers desire to customize the fuzzy sets at the micro level. However, it is unlikely that an end-user would feel comfortable altering these values. We would consider a feasible approach to be an end-user developer/domain expert altering the values in collaboration with a software professional. This can be considered tuning the fuzzy sets to reflect the specific domain it is going to be used in. A caveat with this is the difficulty of fine-tuning a fuzzy set, we consider it to be an incremental process performed over time with only small adjustments made after an initial broad-stroke tuning. We have not evaluated the tuning of fuzzy sets by end-user developers in our pilot evaluation. The fuzzy sets reflect by default a standard setup for a generic domain. The fuzzy sets are used as follows. When a user selects a design pattern, from the design pattern library or common cases, EUREQA performs a simple calculation for each non-functional quality attribute. It subtracts the design pattern's value for the given non-functional quality attribute from the user-defined value for the same attribute. The difference is then used in the fuzzy set membership degree calculation.

Currently the tool supports design pattern selection and deals with presenting the non-functional effects of choosing a design pattern. The tool builds on the promise of model-driven development (MDD), wherein the end-

user can generate UML class diagrams and sequence diagrams along with other platform independent models (PIM) that can then be used as a basis for further refinement in other tools or as the source for code-generation in an MDD environment.

V. Evaluation

This section presents the evaluation phase of the design science process. The next section will describe the data collection. This is followed by the analysis section and finally the results section.

A. Data collection and evaluation method

We recruited three master level information science students to participate in this pilot evaluation. The evaluation was set up with 40-50 minute sessions for each of the three participants, with a final open group discussion lasting approximately 20 minutes. Each participant was initially given a quick demonstration of the tool along with a brief user guide describing the tool's main functions and the problem case. The user guide was to be used for reference when using the tool if the participant at any time felt uncertain or unsure of what to do.

Data capture was done through audio- and screen-recording. We used the think-aloud protocol [39] encouraging participants to talk about their experiences as they were working with the tool. This technique provides "a valuable source of the data about the sequence of events that occur whilst a human subject is solving a problem or performing a cognitive task*"* [40] (p. 10). Ericsson and Simon [41] have shown that the think-aloud protocol does not disturb or detract the participant from the problem-solving task. However, with the think-aloud protocol "there is no room left for reflecting" according to van Someren et al. [39] (p. 26). The observer sat 3 meters behind the participant at a slight angle in order to see the screen. The participants could, if they felt completely at a loss, ask questions. Each participant used the tool for approximately 30 minutes, this was then directly followed by a semi-structured interview with each participant. The same broad topics were discussed with all three participants thus making it possible to synthesize laterally. The evaluation was concluded with a final group discussion. This was to allow the participants to discuss openly in a more reflected manner, something which is not directly supported in the think-aloud protocol. The data has been analyzed using an emergent open coding procedure [39] [42].

B. Analysis

We have previously discussed the concept of a priori constructs and emergent constructs. In Table 1, we show the most important constructs, both a priori and emergent. Constructs marked with a * symbol were defined a priori. Some a priori defined constructs were adjusted during coding, these are marked with a ** symbol. From Table 1 we can see that many of the constructs which were a priori constructs were adjusted. During the coding, we discovered that our descriptive feature constructs were being discussed in relation to how the users understood them. This is reflected in the high number of adjusted a priori sub-constructs in the *Understanding and Comprehension* construct group.

Thus, we had a set of descriptive constructs prior to the data analysis, specifically constructs which were of interest to us, such as *design patterns, user-defined non-functional profile* and *radar chart.* During initial transcribing, raw data analysis and open coding several other central constructs become apparent. We call these emergent constructs, an example of how an emergent construct was created is as follows. For instance the following topically similar statements were all the foundations of emergent constructs.

"I haven't worked with the UML for like three or four years, so please bear with me if I am making a lot of mistakes understanding it" - Participant 1

"*I am not quite up to scratch on my UML and didn't reall y understand how the boxes were related"* - Participant 2

"I just lacked the upfront knowledge about UML diagrams" - Participant 2

"I really feel I need some help here. Because I can't grasp this diagram. Like these arrows, what do they mean?" - Participant 3

The four statements were among the foundations of the emergent construct *background knowledge* and *UML model understanding*. That these became important constructs was interesting since the hypothesis was that using design patterns would allow one to abstract away the details of the UML models, thus reducing the amount of background knowledge required in order to develop a UML model solution.

The first phase of the analysis was counting how many times each construct was used in the coding effort. After the open coding was completed the various a priori and emergent constructs were assembled in exclusive sets.
In total there are seven main construct that contain 28 sub-constructs with a reference count from 1 to 53. The results are presented in Table 1.

TABLE I.
MAIN CONSTRUCTS AND SUB-CONSTRUCTS

Construct	Sub-constructs	Construct frequency and percentage	
Decision Making	Non-functional value tweaking**	10	2.45%
	Design Pattern Selection*	12	2.94%
Rationale	Class Diagram changes reasoning	14	3.43%
	Design Pattern selection reasoning	16	3.93%
	Non-functional quality value tweaking reasoning	18	4.42%
Understanding and comprehension	Case problem understanding**	18	4.42%
	Design pattern understanding**	47	11.5%
	GUI understanding**	15	3.68%
	Non-functional qualities understanding**	24	5.89%
	RadarChart understanding**	10	2.45%
	UML model understanding**	35	8.59%
Negative Experience	Cognitive Overload	7	1.71%
	Confusion	32	7.86%
	Feeling of being restricted	2	0.49%
	Forced Fit	4	0.98%
	Sense of being lost or uninformed	18	4.42%
	Time consuming	3	0.73%
	Unfamiliar action, not the normal way of doing things	3	0.73%
	Unintended use of tool	2	0.49%
Positive Experience	Positive experience	3	0.73%
	Sense of control	12	2.94%
	Well reasoned decision	9	2.21%
Retrospective	Personal preference	5	1.22%
	Preferred approach	27	6.63%
	Reflection	53	13%
	Supporting different user profiles	2	0.49%
	Background knowledge	1	0.24%
Tool functionality	Audio recording*	5	1.22%
	Common Cases*	1	0.24%

Table 1 provides a high number of coding occurences for some of the *negative experience* sub-constructs and similarly for the *understanding and comprehension* sub-constructs. We used NVIVO 8 for our qualitative analysis. A feature of the analysis tool is what is called a matrix coding query. It allows us to analyze the relations between the *negative experience* sub-constructs and the *understanding and comprehension* sub-constructs. A relation exists whenever coding of both a construct from the *negative experience* constructs AND a construct from the *understanding and comprehension* constructs exists.

C. Results

We present the results along two dimensions by using a dichotomous separation between negative and positive coded constructs. The following section presents the negative constructs and their relations to other constructs. Fig. 5 shows the results in a 3-axis column chart.

In Fig. 5, we can see that there are few negative experiences associated with the *Radarchart understanding* for displaying the fit between the user-defined non-functional profile and design pattern's non-functional profile. The most obvious negatively coded issues relate to *Design Pattern understanding* and *UML model understanding*. There exists a relation between the sub-constructs *sense of being lost or uninformed* along with *confusion* and the sub-constructs *UML model understanding* and *Design Pattern understanding*. There is a parallell between these two spikes in Fig. 5, the purpose of design patterns was to abstract away the need for detailed knowledge of the semantics of UML and class diagram modeling. However when this abstraction fails it requires the end-user developer to rely more on the UML model which causes increased confusion or sense of being lost or uninformed. Moreover it became clear that from a task process progression point of view the participants struggled when they reached the class diagram phase. Participant 3 upon reaching the class diagram phase (Fig. 3) commented that “I really do not know where to even start” and during the semi-structured interview debriefing stated that “when I got to this step here [the class diagram] I didn’t know what to do at all” and “I didn't feel I could get much help at all. I tried readi ng the description but it wasn't very useful”. Participant 1 echoed the sentiment of Participant 3 stating that “If I had been more familiar with UML this would have gone much quicker. It is such a long time since I have worked with this”. This reveals that they both immediately started focusing on the semantics and syntax of the UML which ideally would not be a focal area if they could continue thinking at the abstract level of design patterns and the various design pattern roles.

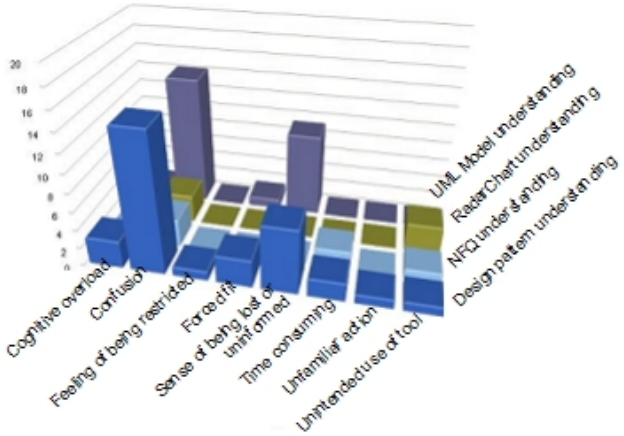

Figure 5. 3-axis column chart showing relations between coding of the Negative Experiences sub-constructs and the Understanding and Comprehension sub-constructs

In regards to the design pattern understanding, one participant noted that the design pattern descriptions came across as "a wall of text" rendering the participant overwhelmed. This view was not shared among all participants. Another participant stated that the design pattern text was quite helpful once he had read it thoroughly. But the participant did note that briefer "at a glance" descriptions would be useful. The disposition of the text descriptions could to a certain extent explain why the anticipated level of abstraction through design patterns was not achieved.

If we consider a similar diagram for the set of positive experiences with the set of understanding and comprehension, in Fig. 6, then we see that the two constructs of *NFQ-understanding* and *RadarChart understanding* sustain spikes. During both the think-aloud part of the evaluation and the following interview, all the participants noted the intuitive approach accommodated by the radar chart.

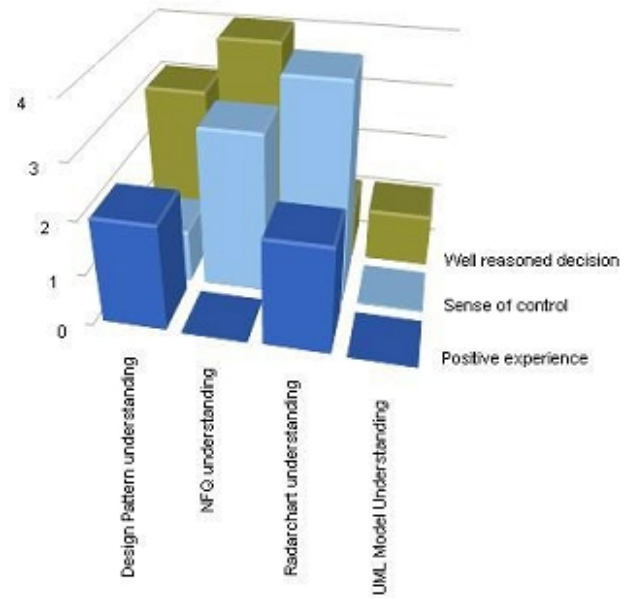

Figure 6. 3-axis column chart showing the relations between the sub-constructs of positive experiences and the sub-constructs of comprehension.

The ease of use associated with the radar chart acted as a catalyst for the high number of *well-reasoned decisions* and *NFQ understanding* codings. There are also similarities between the spike of *NFQ understanding* and *Sense of Control* compared to the spike of *NFQ RadarChart understanding* and *Sense of Control.* Given that the participants did not express any *positive experience* related to *NFQ understanding,* it is reasonable to argue that their experiences with the radar chart gave them more confidence in their NFQ choices. This also trickles through to the well-reasoned decisions related to design patterns since the radar chart provides a graphical bridge between design patterns and non-functional qualities. This bridge allows an EUD to directly assess the suitability of a design pattern.

VI. Discussion

Our results suggest that some of the features such as the radar chart and use of fuzzy sets can be useful. Granted this was only a pilot evaluation with few participants, but as early results they are encouraging. Similarly the participants acknowledged that non-functional requirements were not something they considered in their day-to-day development efforts. They also commented that they liked the concept of being able to numerically create a cohesive profile that acts as constraints on their design choices.

EUREQA is an environment that combines both visual/pictorial and textual information. However, the programming form in EUREQA is visual. The debate as to whether or not visual programming is by default a superior form of programming has been ongoing for decades [43]. An issue is the lack of empirical evidence by those who claim natural superiority of visual languages [44]. Visual programming can be beneficial depending on the context. Arguably the use of the radar chart to create a coherent visual profile of the non-functional requirements and allowing overlays is valuable. The radar chart is used in the tool to make it cognitively easier to comprehend how a design pattern relates to the user-defined non-functional profile. It allows for an immediate impact assessment of *n* selected design patterns versus the user-defined profile, and thus acts as a high-level primitive in the modeling space. In our results the participants were positive to the radar chart visualizations.

Beringer [45] identifies several issues that need to be addressed in order to realize EUD such as "*Intelligent System, High-level semantic building blocks*" and "*Metaphors*" [45] (p. 40). The tool offers support for all these aspects through the use of fuzzy sets, radar charts accompanied by GRL models and Design Patterns respectively. The fuzzy sets give a sense of intelligence in the form of soft computing by allowing for vagueness and degrees of appropriateness when selecting design patterns and matching them against the non-functional profile. Work by Mussbacher, Weiss, and Amyot [46] investigated using URN (which the GRL is a part of) as a means of formalising design patterns and assisting in selecting between design pattern-based solutions.

Moreover their work allows for depiction of the system-wide impact design patterns have on non-functional qualities. Our work uses the same epistemological foundation, but rather than formalizing design patterns using URN we leverage the GRL as a view of how the design pattern selection affects the non-functional constraints. Our evaluation showed that the participants did not find much value in the GRL view, and preferred using the radar chart to assess the fit between their selected design pattern and non-functional profile. However, we believe the value of GRL might only become apparent for users when they can combine multiple patterns. This is because the GRL view can show in more detail the overall impact each pattern has on the non-functional quality attributes of the system.

The most interesting finding was that they struggled with the UML class diagram. One reason for this can be due to the participants not having created the diagram from scratch, as they are used to. This creates a gap in their mental mapping of domain concepts to class diagram elements. Our findings are congruent with findings of Jalil and Noah [47], who observed that novices had trouble mapping from domain pattern roles to class diagram elements. The users did manage to produce partially correct mappings after a while but they were not certain about them. Moreover, gentle slope has been promoted as pivotal in end-user development environments. If EUREQA provided this, we should expect an even distribution of negative experiences across all steps, however instead we see that most of the negative experiences are clustered around activities that take place in the last step. As such EUREQA does not provide a gentle slope through all the steps by introducing a gap in complexity moving from the second to the third, and final, step.

As such, concerns may be raised against the use of UML as the only abstraction level for working with design patterns. Although, in this pilot evaluation version the participants could only create one design pattern in model instantiated form and augment it by manually adding new classes, the question of scalability is pertinent. Burnett et al. [48] discuss the issue of scaling up visual languages. A concern in EUREQA would be the cognitive overload which would occur as a class diagram grows over time during further development. Research by Yusuf, Kagdi and Maletic [49] used eye-tracking equipment to show that providing additional semantic information could aid in understanding growing class diagrams. Currently EUREQA does not provide that visually, but the user can add audio annotations. These could be incorporated as visual cues in the class diagram elements showing which classes have audio annotations attached.

Comparing our results to some of the properties of the cognitive dimensions framework of Green and Petre [20] for visual languages we observe that EUREQA provides an environment supporting *closeness of mapping.* This property delineates the effort of mapping from the problem world to the computer world. However we expected the closeness of mapping to be even more efficient. Although the participants did after a while manage to map the design pattern roles to the classes, it required a lot of effort and they leveraged their previous experience from similar tasks to accomplish it. This is against the design goal of EUREQA allowing even novices to perform this task with relative ease. Another property of Green and Petre [20] is error-proneness, how easy it is to make mistakes in the visual language. The radar chart was useful in the sense that it makes it impossible to make any errors, along with floor and ceiling limits to the acceptable values when defining the user's non-functional profile. However, in the UML class diagram – even if exclusively using generated design patterns to erradicate any syntactical errors, the participants struggled with both the class and connector notational semantics. In EUREQA they should not really be of a concern as the primary task in the class diagram is mapping of roles to classes, but we found the participants fretting over the semantics of the UML at a very basic level. This is a concern because, as one participant mentioned, he couldn't even grasp what the boxes represented. As such there is a high degree of error-proneness if the basic conceptual understanding of how a class relates to the design pattern is lost. This continues on to the next property discussed by Green and Petre [20], namely *role-expresiveness* which pertains to how difficult is it to answer "what is this bit for?*"* (p. 31). The results suggested that the users felt it was clear to them what they were supposed to do in the Main and Common Cases tab. They understood the logic of the activities in those two tabs. However, this faded at the class diagram tab. Arguably a lack of additional information, or what Green and Petre [20] call *secondary notation and escape from formalism,* augmented the issue. Both comparing our results to other empirical results and to theoretical frameworks we see that EUREQA is promising with design patterns existing at a fuitful level of abstraction and the visualization with the RadarChart being considered informative. But there is an issue with lack of information supporting the transition to the class diagram tab.

Of other current end-user development approaches, the component-based one is the most similar to EUREQA's use of design patterns. They both exist at an intermediate level of abstraction and provide finished templates in the form of components and design patterns respectively. A benefit of the component-based approach is that the user works in a strictly managed environment in terms of the component repository and connectivity between components. Moreover component compositions are directly executable allowing the end-user to test a running solution immediately. In EUREQA the user works at a lower level by being given access to the individual classes of the design patterns. It does not sustain the same black-box quality as the component-based approach. This allows end-user developers to incorporate their domain knowledge. However, EUREQA does not allow for immediate execution of a designed artifact. This can be considered both a positive and a negative issue. On one hand being able to create a

running artifact is important in end-user development. Won, et al. found that a barrier to end-user development was that end-user developers were afraid of making changes that would break a run-time solution [18]. In EUREQA this can be managed since a) the end-user only creates models that need to be forward-engineered to create a running solution, and b) during forward-engineering, professional developers can be involved for quality assurance purposes.

We argue our use of a priori constructs coupled with the user guide strengthened the internal validity of the results. The user guide used the same terms to present the features of the EUREQA tool as the a priori constructs. The user guide served two purposes. The first purpose was being a guide and reference for the participant. The second purpose was to establish a feature-specific vocabulary with the participant. The user guide used terminology derived from a priori constructs. This led to very precise feature-specific statements from the participants increasing the validity of the analysis results. We could have further improved the validity of the results by introducing questionnaires to control for any bias introduced in the semi-structured interviews. However, with such a small sample the questionnaires would not have yielded any valuable statistical results.

One obvious limitation of this work is the small number of participants in the evaluation. This reduces the external validity and generalizability of the results. Another external validity threat is that the participants were not actual end-user developers. First of all they are more experienced than an average end-user developer. Secondly since they do not work in an organization related to the case problem they have no vested interest in the case problem or domain knowledge. However this is a pilot evaluation and our research goal at this stage is to assess opportunities and challenges with EUREQA rather than empirical generalizable results.

VII. Conclusions And Further Work

End-user development has been touted as the "holy grail for tool developers" by Sutcliffe, Lee and Mehandjiev [50] and there is little doubt that end-user development will become a more important and feasible software development approach as more and more companies move towards frameworks and middleware solutions for increased flexibility and extendability. One of the main barriers to end-user development is the cognitive complexity of entering into the realm of coding and software development [51]. Although we have come along way with 4th generation programming languages we are still short of delivering on the promise of *accessible* model driven development. End-user developers need to work at the model level with a high level of abstraction closely tied to their real-world domain [19]. This paper has introduced a first versioin of the EUREQA tool for EUD using design patterns as the point of departure from which UML model design solutions are built. The tool was evaluated using a qualitative approach based upon the think-aloud protocol as participants attempted to solve a real-world case using the tool. The evaluation disclosed that at the tool level there are several usability issues that need attention. However, of greater interest, the evaluation also indicated that the concepts and premise of using novel graphical presentation techniques for design solution choices focusing on non-functional qualities worked well, and design patterns proved, although less than expected, to provide abstraction and suffice as a plateau from which end-user developers could select viable solutions.

The challenge of producing end-user developed quality software that precisely meets business requirements is still not resolved. This work shows that using visualization techniques can help end-user developers in dealing with non-functional requirements. The results also show that design patterns can aid in reducing the cognitive gap between the domain- and computer-model. But this requires high-level descriptions besides the problem-, solution-, context-, and intent descriptions of design patterns.

Further work on EUREQA will involve additional evaluations with a higher number of participants, and incorporate a wider range of data collection techniques, such as questionnaires. Our findings and data from this evaluation will be used as valuable input to redesign in accordance with the design science method. The most urgent issue is improving the transition and work in the final class diagram modeling step. As stated in the previous section, EUREQA must also be extended to handle automated composition of multiple design patterns.

A different area is investigating the use of analytic hierarchy process (AHP) as a means of comparing and selecting between design patterns that only differ in their non-functional profile. An AHP approach would be robust in that it would allow us to capture both objective and subjective evaluation measurements.

Currently, EUREQA only allows for a single-stakeholder view, where the GRL allows modeling of multiple stakeholders. This is a feature we envisage could be valuable in an EUD-setting and will be investigated in our further work.

A final aspect aspect are the values set for the non-functional properties of the design patterns. Currently, they are derived through subjective consideration. This does detract from their validity. In a pure pilot evaluation for usability testing, we argue this is acceptable, however if used for real-world development, it would be necessary to use a formal and verifiable approach to set the design pattern non-functional profile values such as the techniques proposed by Hsueh and Shen [28] or Fletcher and Cleland-Huang [30].

Acknowledgements

This work has been done using a grant by the Norwegian Research Council on VERDIKT project no. 10280903.

REFERENCES

[1] C. Scaffidi, M. Shaw, B. Myers. *Estimating the Numbers of End Users and End User Programmers* IEEE Symp. on Visual Languages and Human-Centric Computing. Dallas, TX, USA. September 2005.

[2] D. Buck, D.J. Stucki. *JKarelRobot: a case study in supporting levels of cognitive development in the computer science curriculum*. Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education. Charlotte, NC, USA. February 2001.

[3] M.F. Costabile, P. Mussio, L.P. Provenza, A. Piccinno. *End users as unwitting software developers*. Proceedings of the 4th international workshop on End-user software engineering. Vancouver, Canada. May 2008.

[4] M. Ancona, G. Dodero, F. Minuto, M. Guida, V. Gianuzzi. *Mobile computing in a hospital: the WARD-IN-HAND project*. Proceedings of the 2000 ACM symposium on Applied computing - Volume 2.

[5] Rugge, C. Ruthenbeck, J. Piotrowski, C. Meinecke, F. Bose. Supporting mobile work processes in logistics with wearable computing. Proceedings of the 11th International Conference on Human-Computer Interaction with Mobile Devices and Services. Bonn, Germany. September 2009.

[6] T. Nicolai, T, T. Sindt, H. Kenn, H. Witt. *Case Study of wearable Computing for Aircraft Maintenance.* Proceedings of International Forum on Applied Wearable Computing IFAWC'05, Zurich, Switzerland. March 2005.

[7] J. Krogstie, K. Lyytinen, A.L. Opdahl, B. Pernici, K. Siau, K. Smolander. Research areas and challenges for mobile information systems. *International Journal of Mobile Communications, 2*(3), (2004) 220-234.

[8] H. Lieberman, F. Paterno, M. Klann, M. Wulf. End- User Development: An Emerging Paradigm. In H. Lieberman, F. Paterno & V. Wulf (Eds.), *End User Development*. AA Dordrecht, The Netherlands: Springer. (2006)

[9] E.L. Wagner, G. Piccoli. Moving beyond user participation to achieve successful IS design. Communications of the ACM, 50(12), (2007) 6.

[10] T. Klaus, S. Wingreen, J.E. Blanton. Examining user resistance and management strategies in enterprise system implementations. Proceedings of the 2007 ACM SIGMIS CPR conference on Computer personnel research: The global information technology workforce. St. Louis, MO, USA. April 2007.

[11] R. Hirschheim, M. Newman. Information Systemand User Resistance: Theory and Practice. The Computer Journal, 31(5), (1988) 398-408.

[12] G. Fischer, E. Giaccardi, Y. Ye, A.G. Sutcliffe, N. Mehandjiev. Meta-Design; A Manifesto for End- User Development. Communications of the ACM, 47(9), (2004) 5.

[13] J. Segal. (2007). Some Problems of Professional End User Developers. IEEE Symposium on Visual Languages and Human-Centric Computing. VL/HCC 2007. Coeur d'Alène, Idaho, USA. September 2007.

[14] L. Chung, J. d. P. Leite. On Non-Functional Requirements in Software Engineering. In A. T. Borgida, V. Chaudhri, P. Giorgini & E. Yu (Eds.), Conceptual Modeling: Foundations and Applications (pp. 363-379). Berlin / Heidelberg: Springer. (2009)

[15] M. Kassab. Non-Functional Requirements: Modeling and Assesment. Saarbrucken, Germany: VDM Verlag, Dr. Müller. (2009)

[16] N. Mehandjiev, A. Sutcliffe, D. Lee. Organizational view of End-User Development. In H. Lieberman, F. Pateró & V. Wulf (Eds.), End-user Development (pp. 492): Springer. (2006)

[17] Mørch, G. Stevens, M. Won, M. Klann, Y. Dittrich, V. Wulf. Component-based technologies for end-user development. Commun. ACM, 47(9) (2004) 59-62.

[18] M. Won, O. Stiemerling, V. Wulf. Component-based Approaches to Tailorable Systems. In H. Lieberman, Paternò, F., Wulf, V. (Eds.), End-User Development (pp. 115-141). Dordrecht, NL: Springer. (2006)

[19] J.F. Pane, B. Myers. More Natural Programming Languages. In H. Lieberman, Paternò, F., Wulf, V. (Eds.), End-User Development (pp. 31-50). Dordrecht, The Netherlands: Springer Verlag. (2006)

[20] T. R. G. Green, M. Petre. Usability Analysis of Visual Programming Environments: A Cognitive Dimensions' Framework. Journal of Visual Languages & Computing, 7(2) (1996) 131-174.

[21] PERS, (2006, 28th of April, 2006). PERS End-User Development (EUD) Standards. Retrieved 5/9, 2010, from http://www.oregon.gov/DAS/EISPD/ESO/SecPlan/PERS/End_User_devstandards.pdf?ga=t

[22] E. Gamma, R. Helm, R. Johnson, J.M. Vlissides. Design Patterns: Elements of Reusable Object- Oriented Software: Addison-Wesley Professional. (1994)

[23] ISO/IEC Standard 9126-1 Software Engineering – Product Quality – Part 1: Quality Model, 2001.

[24] M. Glinz *On Non-Functional Requirements.* Proceedings of the 15th International Requirements Engineering Conference, RE'07. New Delhi, India. October 2007.

[25] D. Gross, E. Yu. From Non-Functional Requirements to Design through Patterns. *Requirements Engineering, 6*(1), (2001) 18.

[26] L. Chung, B.A. Nixon, E. Yu, J. Mylopolous. *Non-Functional Requirements in Software Engineering* (1st edition ed.): Springer. (1999)

[27] J. Cleland-Huang. *Toward improved traceability of non-functional requirements*. Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering. Long Beach, CA, USA. November 2005.

[28] N.-L. Hsueh, W.-H. Shen. *Handling Nonfunctional and Conflicting Requirements with Design Patterns*. Proceedings of the 11th Asia-Pacific Software Engineering Conference. Busan, Korea. November 2004.

[29] J.Cleland-Huang, D. Schmelzer. *Dynamically Tracing Non-Functional Requirements through Design Pattern Invariants*. Workshop on Traceability in Emerging Forms of Software Engineering, in conjunction with IEEE International Conference on Automated Software Engineering. Montreal, Canada. October 2003.

[30] J. Fletcher, J. Cleland-Huang. Automated Generation of UML Class Diagrams from Softgoal Patterns. DePaul CTI Research Symposium / Midwest Software Engineering Conference (CTIRS/MSEC 2006). Chicago, IL, USA. April 2006.

[31] A.R. Hevner, S.T. March, J. Park, S. Ram. Design Science in Information Systems Research. *Management Information Systems Quarterly MISQ, 28*(1), (2004) 30.

[32] S. March, V. Storey. Design Science in the Information Systems Discipline: An Introduction to the Special Issue on Design Science Research. *MIS Quarterly, 32*(4), (2008) 725-730.

[33] S.T. March, G. F. Smith. Design and natural science research on information technology. *Decis. Support Syst.*, *15*(4), (1995) 251-266.

[34] V. Vaishnavi, B. Kuechler. (2007, 4th of August 2007). Design Research in Information Systems. Retrieved 2/2, 2007, from http://www.isworld.org/Researchdesign/drisISworld.htm

[35] K.M. Eisenhardt. Building Theories from Case Study Research. The Academy of Management Review, 14(4), (1989) 532-550.

[36] U. Flick. An Introduction to Qualitative Research. Thousand Oaks, CA: SAGE Publications. (2009)

[37] ITU-T. ITU-T, Recommendation Z.151 (11/08), User Requirements Notation (URN) – Language definition. Retrieved 11/1-2011 http://www.itu.int/rec/T-REC-Z.151/en November 2008.

[38] R.C. Martin. . Visitor. Retrieved 2/11, 2008, from http://staff.cs.utu.fi/~jounsmed/doos_06/material/Visitor.pdf (2002).

[39] M.W. van Someren, Y.F. Barnard, J.A.C. Sandberg. THE THINK ALOUD METHOD: A practical guide to modelling cognitive processes. London, UK: Academic Press. (1994).

[40] K.A. Ericsson, H. Simon. Protocol Analysis. Protocol Analysis Retrieved 19/9-2010, 2010, from http://octopus.library.cmu.edu/cgi-bin/tiff2pdf/simon/box00082/fld06587/bdl0003/doc0001/simon.pdf (1981, 28th of December 1981)

[41] K.A. Ericsson, H. Simon. Protocol Analysis: Verbal Reports as Data (revised edition). Cambridge, Mass: MIT Press. (1993).

[42] C.F. Auerbach, & L.B. Silverstein. Qualitative Data Analysis: an introduction to coding and analysis Available from http://site.ebrary.com/lib/bergen/docDetail.action?docID=10078435 (2003).

[43] N.C. Shu. Visual Programming: John Wiley & Sons, Inc. (1992).

[44] B. Nardi. A Small Matter of Programming: Perspectives on End User Computing. Cambridge, Mass: MIT Press. (1993).

[45] J. Beringer. Reducing Expertise Tension. Commun. ACM, 47(9), (2004) 3.

[46] G. Mussbacher, M. Weiss, D. Amyot. Formalizing Architectural Patterns with the User Requirements Notation. In Taibi, T. (Ed.) Design Pattern Formalization Techniques. (pp. 302-323) Hershey, New York IGI Publishing Group. (2006).

[47] M.A. Jalil, S.A.M. Noah. The Difficulties of Using Design Patterns among Novices: An Exploratory Study. The Fifth International Conference on Computational Science and Applications. Kuala Lumpur, Malaysia. August 2007.

[48] M.M. Burnett, M.J. Baker, C. Bohus, P. Carlson, S. Yang, P. Zee. Scaling Up Visual Programming Languages. Computer, 28(3), (1995) 45-54.

[49] S. Yusuf, H. Kagdi, J.I. Maletic. Assessing the Comprehension of UML Class Diagrams via Eye Tracking. Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC '07), Alberta, Canada. June 2007.

[50] A.G. Sutcliffe, D. Lee, N. Mehandijev. Contributions, costs and prospects for end-user development. Proceedings of the third International Human Computer Interaction Conference. Crete, Greece. June 2003.

[51] Y. Wang, J. Shao. Measurement of the cognitive functional complexity of software. Proceedings of the Second IEEE International Conference on Cognitive Informatics. London, England. August 2003.