User-centric Programming Language

Wiktor Nowakowski

Warsaw University of Technology, Warsaw, Poland
nowakoww@iem.pw.edu.pl

Abstract. Ambiguously formulated and constantly changing requirements for software systems make it hard to translate them into working code. To overcome these problem, we propose an approach that consolidates the requirements specification level with the design and implementation levels in order to shorten the path from initial requirements to the final code. The end-user of the system will be able to specify requirements in a precise, semantically rich and domain independent User-centric Programming Language (UcPL), which will allow for direct transformation into application logic code.

1 Problem Description and Motivation

Typical software development lifecycles comprise activities that lead from the initial requirements to the final working system. These activities produce various artifacts at different level of abstraction: design models and code. The level of technical complexity makes these artifacts inaccessible to the end-users of the system. However, end-users can discuss the logic (functionality) of the system that abstracts away all the technical details of the software system. This should be done in a language understandable to them. The logic of the problem is usually expressed within the user requirements specifications.

Considering this two main problems arise. The first problem pertains to the process of transition from end-user needs into design and implementation. This includes problems with requirements elicitation and translating the functional requirements into the logic of the system. Attempts to solve these problems are mainly based on the introduction of formal and semi-formal requirements specification languages and automation of transformation between requirements and certain technical artifacts.

The second problem is that the path from requirements to code makes it difficult to maintain appropriate traceability links. This is especially an issue in projects that face high changeability of the requirements. Attempts to solve these problems include rapid application development environments, certain agile methodologies, which try to shorten the path from requirements to code and certain generative approaches where the traces are generated automatically.

Despite shifting the level of abstraction through the introduction of the object-oriented paradigm, still, the existing approaches necessitate a significant level of technical expertise to develop a working system. Thus, there is a rising need to bring the end-user closer to the software developers. The idea here is to let the end-users write some significant portions of software systems. The Gartner Group, in its 2008 study predicts significant growth in end-user software development. In 2010 an average of 34% of companies are expected to conduct more than 20% of application development outside of IT. Unfortunately, there are no approaches to allow the users to construct software through writing the problem logic at the level of requirements. The traditional way of producing software systems is to shift from requirements to code in a generally manual process. More modern approaches try to utilize dedicated tools to automate that process. This necessitates formalizing requirements and constraining the natural language in which they are normally written.

2 Related Work

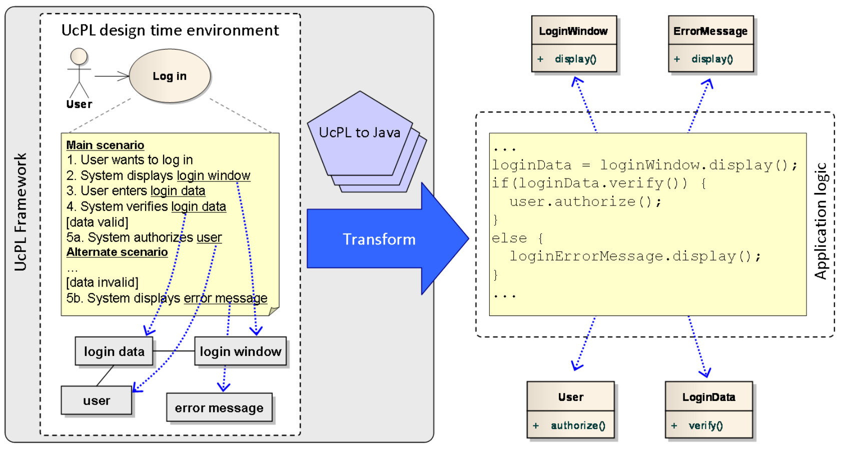
Requirements engineering has been established in the industry as a method for flawless transition from the early system vision over the design, implementation up to the validation and tests. The current practice in this area is more focused on requirements management [1] and requirements interchange [2] than requirements specification. There are also approaches where requirements are specified more formally, like the Four Variable Model [3], [4]. In UML [5] or in its profiles like SysML [6] or MARTE [7], requirements can be handled through creating semi-formally defined visual models. In RSL [8], requirements are specified in constrained natural language with a well defined grammar.

Few of the above mentioned general purpose requirements languages can execute the requirements specification written in it, or can be transformed directly into complete working code. On the other hand, in recent years there has been a drive towards developing executable domain specific languages (DSL) [9]. The idea is that it is more efficient to create a simple language and the corresponding code generators than to apply general purpose languages. The questions are how simple the language has to be and how frequent the language must change to cope with rising demands by its users.

3 Proposed Solution

To go further in resolving mentioned problems, we propose to allow the end-users actively participate in writing software while retaining their role of "requirements specifiers". They would specify the system logic in the language understandable for them and then automatically transform such specifications directly into code. This would certainly lead to significant gains in productivity.

We aim at delivering a framework consisting of three main elements: the User-centric Programming Language (UcPL), the design time environment and the transformation engine along with appropriate transformation algorithms. Figure 1 shows an overview of the UcPL approach. In UcPL we will substitute procedural or object-oriented programming with, so called, user-centric programming. The user-centric language constructs would just allow to specify the logic of the problem. This means that the user-centric programs could be written and read



Main scenario
1. User wants to log in
2. System displays login window
3. User enters login data
4. System verifies login data
[data valid]
5a. System authorizes user
Alternate scenario
...
[data invalid]
5b. System displays error message

```
...
loginData = loginWindow.display();
if(loginData.verify()) {
  user.authorize();
}
else {
  loginErrorMessage.display();
}
...
```

Fig. 1. Overview of UcPL approach

by end-users with no software development background. Most of the design and technological decisions will be hidden by the language platform.

A scenario of developing software application with UcPL is very straightforward. End-users creates requirements specification in the form of user-centric program. Then they choose the desired transformation algorithm. Precision required in order to apply automatic transformation, necessitates extensive mechanism for validation and correction of the UcPL language constructs. Validation should precede the transformation step. The transformation engine takes complete requirements specification in the form of user-centric program as an input and produces complete code of the application logic (or controller in MVC model). The output code lacks implementation of the application's business model – only class stubs with appropriate methods are generated. The application logic code contains calls to these business entities. The same pertains to the user interface elements like windows, buttons, messages, etc. However, the generated code constitutes the application framework which can be compiled and executed, what significantly shorten the time needed to develop the whole system.

In order to generate code of a full system, appropriate extension of UcPL for specifying business logic and user interface would be needed. This is, though, outside the scope of this work.

A programming language (Java, C++, etc.) and all technological details (e.g. use of a specific user interface technology or specific application framework) of the generated code will depend on the transformation algorithm used. The transformation engine built into the UcPL framework will be able to run any transformation algorithm specified in appropriate transformation language.

In order to ensure end-user comprehension, the UcPL language will be based on use case scenarios written in natural language with simple imperative sentences (e.g. "User enters login data“ or "System calculates exchange rate“) and control sentences expressing conditions, loops, etc. (e.g. "exchange rate greater than previous exchange rate").

The language will clearly separate description of the user-system interaction from the description of the domain (see Figure 1). Scenarios will be hyperlinked with appropriate notions defined in a separate vocabulary. Such hyperlinks could be then transformed into operation calls from application logic to business logic and user interface layer.

The syntax of the language will be precisely defined as a meta-model in MOF in order to enable automatic handling of user-centric programs.

4 Research Method

As we mentioned UcPL will need to combine informality with necessary precision, that the end-users would be able to comprehend and write user-centric programs, as well as they are able to understand and write common-prose requirements. The language that addresses most of mentioned issues is the RSL language [10] which was recently developed as a part of the ReDSeeDS project [11]. RSL allows for transformation from requirements specification into draft model of system's architecture. UcPL will be based on the RSL which will be additionally formalized by specifying precise semantics for all the language constructs.

Also an appropriate set of transformation algorithms will be implemented in the model transformation language MOLA [12]. Though there are many transformation languages like QVT [13] or ATL [14], MOLA is preferred for its readability. MOLA is a graphical transformation language where an advanced pattern mechanism is combined with simple traditional control structures. Moreover, MOLA offers comprehensive transformation engine.

An important part of the solution is the design time environment for writing user-centric programs and performing transformations to code. It will be implemented as an extension to the existing ReDSeeDS tool which offers appropriate infrastructure that can be utilized by using plug-in mechanism.

This will enable us to build a system that allows for developing software applications by the end-users by direct transformation from requirements to application logic code. Preliminary studies shows that the proposed approach is possible through skilful extension and combination of the existing technologies.

This goal of retaining end-user comprehensibility of the UcPL as well as useability and effectiveness of the whole framework will be assured and validated through extensive experimental studies. These studies will be mainly carried out by students during software engineering courses. The students will be divided into two subgroups. One group will be developing a system using UcPL approach while the second group will be developing the same or similar system in a traditional way. The results will be compared and analyzed taking into account such factors as time needed to develop the final system and quality of the system measured as the degree of initial user requirements fulfillment.

References

1. Leffingwell, D., Widrig, D.: Managing Software Requirements: A Use Case Approach, Second Edition. Addison-Wesley (2003)
2. ProSTEP: Requirements interchange format (rif). Technical Report PSI 6 Version 1.2, ProSTEP iViP (2009)
3. Parnas, D.: Systematic documentation of requirements. Requirements Engineering, IEEE International Conference on **0** (2001) 0248
4. Heitmeyer, C., Archer, M., Bharadwaj, R., Jeffords, R.: Tools for constructing requirements specifications: The scr toolset at the age of ten. Computer System Science and Engineering Journal **vol. 1** (2005) 19 – 35
5. Object Management Group: Unified Modeling Language: Superstructure, version 2.2, formal/09-02-02. (2009)
6. OMG: Sysml -omg systems modeling language. Technical report (2008)
7. OMG: Marte - uml profile for modeling and analysis of real-time and embedded systems. Technical report (2008)
8. Śmiałek, M., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T.: Introducing a unified requirements specification language. In Madeyski, L., Ochodek, M., Weiss, D., Zendulka, J., eds.: Proc. CEE-SET'2007, Software Engineering in Progress, Nakom (2007) 172–183
9. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons, Inc. (2008)
10. Kaindl, H., Śmiałek, M., et al.: Requirements specification language definition. Project Deliverable D2.4.1, ReDSeeDS Project (2007) www.redseeds.eu.
11. Śmiałek, M., Kalnins, A., Ambroziewicz, A., Straszak, T., Wolter, K.: Comprehensive system for systematic case-driven software reuse. Lecture Notes in Computer Science **5901** (2010) 697–708 SOFSEM'10.
12. Kalnins, A., Barzdins, J., Celms, E.: Model transformation language MOLA. Lecture Notes in Computer Science **3599** (2004) 14–28
13. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, version 1.0, formal/08-04-03. (2008)
14. Jouault, F., Kurtev, I.: Transforming models with the ATL. Lecture Notes in Computer Science **3844** (2005) 128–138