Application Languages: A Product of Requirements Analysis

Julio Cesar S. P. Leite
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
R. Marquês de S. Vicente 225 Rio de Janeiro 22453
Brasil

January 1989

Abstract

Requirements modeling demands that the knowledge of what should be modeled be available. Acquiring this knowledge or eliciting the necessary requirements is recognized to be a very hard problem. One of the aspects that make this task so difficult is the different social environments of the actors involved in the process. Usually, the literature addresses this problem by talking about users and developers. Although the problem is much more complex, this first cut vision does help in pointing out the problem. Modeling the requirements, after they are elicited, has been a favorite topic for researchers in software engineering, which have been proposing a series of different representations. Application languages is another of those proposals, but, differently, this approach aims at the crucial problem of the different social backgrounds of the actors involved in the requirements analysis process.

1 Introduction

We have studied [Freeman 86] the problem of requirements analysis using a process oriented definition. According to this definition, requirements analysis is seen as composed of two processes: requirements elicitation and modeling (see Figure 1).
We further [Leite 88] [Leite 89] studied the requirements elicitation problem, in special
the early validation of requirements using a viewpoint resolution strategy. In studying the
requirements analysis as a process, where different actors having different views try to produce
a requirements, we were dealing with the crucial problem of the different backgrounds of each
actor. In our research we have focused on a way of validating requirements by comparing
different views, taken from different viewpoints.

Stating that we only have users and developers is, as said before, a simplification of
the problem. In reality the several actors that are classified as users and the several actors
classified as developers do have different viewpoints. The process of resolving those viewpoints
into a reconciled requirements is a difficult task. Since it is a social process, the negotiation
between the actors is not an algorithmic process and needs further empirical research.

Requirements modeling as proposed by software engineers is usually based on a general
high level language, that aims at providing organization and representation capabilities that
should cover a wide spectrum of problems. One of the major problems of those modeling
techniques is that they are biased towards the viewpoints of the developers, and as such are
not considered by users as a good communication vehicle [Colter 84].

Our proposition is based on an anthropological approach to the problem. If we assume
that an application area has its own culture we, then, have two strategies that should be used
during the process of requirements analysis.

- The actors involved in the process that are not familiar with the application—should
  "live it", that is, they should participate for some time with the actors involved in the
  application.

- Since language is a reflection of culture, the process of requirement analysis must include
  the task of capturing the language of the application.

This article focuses on the second strategy, that is, the production of an application
language as the result of the requirements analysis process. In the sections that follow, we report on related work, on the use of semiotic ideas, and on a detailed description of our proposal. We conclude by addressing the possible impacts of such an anthropological approach on the software process, as well as our research plan to further investigate those new ideas.

2 Languages, the Computer Science Viewpoint

Computer science has learned since its early days the importance of languages. The necessity of providing more than just mnemonic hardware level operations, as in the assemblers, started the proposal and development of a series of "high level languages". The development of those languages used the theoretical framework inherited, mostly, from the work of Chomsky on generative grammars. This general framework is still in use today, having most of the efforts been concentrated in developing formal syntax and formal semantics for those languages. The syntax effort resulted in an almost universal notation (BNF), but the semantics has not achieved a consensus in the computer science community.

The common approach used by computer scientists in creating computer languages was the one of avoiding the details of the assemblers and providing a general computational framework, oriented towards some very broad area of application. If we analyze the history of programming languages, we will note that FORTRAN, the predecessor of high level languages, was designed as a very general programming language, biased, of course, by the dominant use of computers as high speed calculators. The use of computers as information processors made necessary to have a language were the suppression of details from the assemblers were oriented towards the general world of information processing, and as such COBOL was developed. Following those two landmarks, hundreds of languages were defined and implemented, always aiming at a broad area of application.

All of those implementations and proposals used a bottom-up philosophy, which is the search for an even more high level language, general enough to solve most of the problems. ADA, Pascal, C, Modula-2 are all recent examples of such an approach. It is also an example of this philosophy the application generators [Horowitz 85], which, although encapsulating more "knowledge" than the previous cited languages, is still aimed at a very broad class of applications. The movement towards specifications languages, although providing less level of detail than programming languages, also follows this bottom-up philosophy [Leite 87].

An interesting instance of programming language that represents a departure, although not a big one, from this philosophy is APL. APL not only tried to avoid the assembler level
of detail, but did capture complex abstractions regarding matrix calculation in sacrifice of a wide range of applicability. As a result, the language has been very popular with professionals using a lot of matrix and vectorial calculations, but with almost no acceptability with the overall computing community.

Bentley [Bentley 86] has pointed out the advantages of using linguistic insights in his "programming pearls" section on the CACM. He says:

"Languages surround programmers, yet many programmers don't exploit linguistic insights. Examining programs under a linguistic light can give you a better understanding of the tools you now use, and can teach you design principles for building elegant interfaces to your future programs."

The use of very restricted languages, called little languages, for very restricted problems domains is advocated by Bentley as a way of achieving a better design, as well as the possibility of future reuse. Although still having the strong influence of the work on translation (bottom-up), characteristic of programming languages, little languages is a departure from the wide-spectrum philosophy of traditional programming languages. Even though restricted to the social arena of the "developers", it uses the principle of encapsulating "knowledge" for a specific task in the form of a language. Bentley cites some examples on his article, and [Bos 88] in a different context, gives an example of such an approach.

A considerable departure from the bottom-up philosophy is the work of Neighbors [Neighbors 84] with the Draco paradigm [Arango 88a]. For Neighbors, languages should reflect an application domain, independently if the domain is in the social arena of the "developers" or in the social arena of "users". Specification would be written using a network of languages, each one being the one suited for the particular domain in question. Neighbors believed that a process called Domain Analysis and a process called Domain Design should produce a language for a specific domain. Using this framework Neighbors showed how it is possible to construct software by assembling components, and as a consequence to achieve reuse of analysis and design.

Since domain design, that is, the construction of a language, with its semantics and syntax is not an easy task and because there was a goal of achieving reusability of analysis. Neighbors believed that a domain analysis have had to be performed before producing a language. It happens that domain analysis is a difficult process, with few insights of how it should be done [Arango 88b]. As such, the situation brings itself to a sort of catch-22, that is, there is a need of a domain analysis to justify a domain design and the task of domain analysis is not yet well understood. The important point to stress however is the idea of using the language of the problem to express it.
In the next section we summarize some of the basic ideas of semiotics and use some of those ideas in proposing an overall schema for the use and production of application languages.

3 Application Languages

Eco [Eco 79] believes that it is necessary to distinguish between \textit{signification} and \textit{communication}, where the process of communication needs a system of significations. Although recognizing that they are strictly intertwined, Eco proposes a theory of codes and a theory of sign production. If a system of codes should reflect a previously existing social convention, then it is obvious that one should capture this "social convention" before producing the system of codes, but this system of codes must follow the social convention if it wishes to be a medium of communication in the same social setting.

Using Eco's theory of codes, signs are seen as \textit{sign-functions}. "A sign-function arises when an expression is correlated to a content, both the correlated elements being the functives of such a correlation". A system of codes is the underlining representation used by sign-functions. Eco believes that a system of codes has four different entities:

- signs,
- notions,
- behavioral responses, and
- rule.

It is clear that the similarity with generative grammars is not a coincidence. The important point is that what should be present in a system of codes, and that system be available before the specification of a software system. We are calling \textit{application language} this system of codes related to a certain application. Although calling it a language, we are not strictly following the computer science tradition of using a generative grammar model.

Signs are the syntactic representations, the notions are the set of contents that could be related to a sign depending on different viewpoints. Expressions or signs could be related to different contents or notions, as well as contents or notions could be related to different signs.

\footnote{Eco distinguishes between signal and sign. Signal if there is no concern with its content. Since we use signal with a semiotic purpose, we use sign for meaning sign and signal.}
Behavioral response is the effect of a sign-function on the universe of discourse². Rule is the set of relationships linking signs to notions and to behavioral responses.

Using the library system described in the IWSSD [Iwssd 87], a possible partial description of its application language would have the following format:

<table>
<thead>
<tr>
<th>sign:</th>
<th>(1) borrow a book.</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign:</td>
<td>(2) check-out a book.</td>
</tr>
<tr>
<td>sign:</td>
<td>(3) return a book.</td>
</tr>
<tr>
<td>notion:</td>
<td>(4) a user of a library is the temporary owner of &quot;a book&quot;.</td>
</tr>
<tr>
<td>notion:</td>
<td>(5) a user of a library relinquishes to the library the temporary ownership of &quot;a book&quot;.</td>
</tr>
<tr>
<td>response:</td>
<td>(6) library does not have the copy of &quot;a book&quot; temporarily, and the user has &quot;a book&quot; temporarily.</td>
</tr>
<tr>
<td>response:</td>
<td>(7) user does not the copy of &quot;a book&quot; anymore, the library has the copy of &quot;a book&quot;.</td>
</tr>
<tr>
<td>rule:</td>
<td>(1) --&gt; (4) --&gt; (6), (2) --&gt; (4) --&gt; (6), (3) --&gt; (5) --&gt; (7).</td>
</tr>
</tbody>
</table>

The above example needs two explanations. First, only one viewpoint was considered. Second, some would claim that there is a lack of formalization of the real semantics, but that is exactly what an application language should reveal. The notions and the responses assigned to a given sign should be enough given an existing social convention. That is the point of using this semiotic approach.

It is clear however, that it may be necessary to have other viewpoints represented in the language. As an example, we could have the physical viewpoint where the notions and the responses should convey the content as regarding physical quantities, location and the related information.

Another viewpoint could be that of a control system that keeps track of who the temporary owner is, the name of the book, and for how long the user has the temporary ownership.

The important point to stress is the importance of the concept of “application vocabulary” [Freeman 88] where the signs should maintain the same syntactic form as it exists in the culture being observed. The sign, in an established culture, is important because it carries not only the expression as well as the content. Although this is true, when mapping the system of codes of an application, one can not excuse himself of providing the sign, the notion, the response and the rule.

²Universe of discourse is the overall context in which the software will be developed. The universe of discourse includes all the sources of information and all the people related to the software. These people are referred to as the actors in this universe of discourse. It is the reality trimmed by the set of objectives established by the ones demanding a software.
In this proposition we used semiotics as to provide a system of codes for representing the signification part. It is obvious that our scheme has to be further developed to support the communication part as well. The communication part would be the expression of the requirements itself. The application language is the underlining representation for that expression.

Here we come to a point where there is a need to explain the difference of our approach from the one put forward by Neighbors in the Draco framework. Neighbors believes a language should be elicited as a result of a domain analysis. Neighbors calls a language a "formal" representation using a generative grammar, where syntax is defined via a BNF and semantics are formalized by components, which ultimately should have an operational definition (an executable language). The process of constructing this language needs the expertise of a domain designer. On the other hand, we are proposing a system of codes (a language) free of implementation considerations. We are proposing a language as the necessary medium in which the requirements should be expressed. So we are using the idea that "the laws of signification are the laws of culture" [Eco 79], but neither inheriting the burden of domain analysis nor of the domain design.

Next, we briefly sketch possible impacts of using application languages as the product of requirements analysis in the process of software, as well as examine the relevance of our research regarding the state of the art in software engineering.

4 Conclusion

As of now we are starting to develop several application languages, in order to better evaluate the methods, most of them based on our previous work on viewpoints [Leite 88], and to implement a semantic net for "rule", that is, a semantic net linking the signs, notions and responses in an organized way as to provide storage and retrieval capabilities.

This anthropological approach, that uses semiotics as a tool to talk about the culture of an application, has the following implications on the software process.

- Although very similar to the idea put forward in Draco [Neighbors 84], it does assume that there is no need for domain analysis. Each application problem has its own language depending on the local culture of where it exists. Nonetheless, it may happen that an application language may export some of its sign-function to other instances of similar application languages. As proposed, the system of codes is easy to be maintained, as well as to export sign-functions, but the work on its implementation, as a semantic net, will give further insights in the problem of requirements analysis reusability.
In expressing notions, it is possible and desirable to have the contents expressed as signs of other application languages. This network of languages makes it possible a high level of abstraction via the reuse of previous analyzed applications.

Because it does not have a rigid representation, but has the essential information necessary to a formalization, it is possible, once found a suitable representation, to have an early formalization that could lead to an automatic treatment. In particular, it seems possible to have a scheme similar to the one proposed in Draco, that is, having a transformation based system to support a system with a network of languages.

Since the application language has the laws of signification of a given application, it has the culture of that application. Having nailed down a culture of an application we are attacking the crucial problem of communication between user and developer.

The phase of requirements analysis will certainly cost more, since there will be a need for expliciting a system of codes, before stating the requirements using this system of codes. No figures of cost evaluation are available, but it is one of our research plans to measure the impact of producing the application language versus the normal requirements analysis methods.

We firmly believe that the idea of “application languages” is essential to obtain a high level of abstraction at the requirements analysis process without relinquishing the readability and comprehension by the actors, which belong to the social arena called the users. Eliciting information and presenting it according to the culture of users, is essential to the process of software development.

References


