Software Reuse Research at KSLA

Ben A. Sijtsma

Koninklijke/Shell–Laboratorium, Amsterdam
(Shell Research B.V.)
P.O. Box 3003, 1003 AA Amsterdam, The Netherlands
e-mail: SIJTSMA1@KSLA.NL

Abstract

This paper briefly discusses the research efforts aimed at providing reuse technology for the Laboratory. Some characteristic properties of software development at the Laboratory are considered and their influence on software reuse are presented. We conclude with an outline of our approach to domain analysis.

Keywords: repository construction, domain analysis

Introduction

The research on software reuse at Koninklijke/Shell–Laboratorium, Amsterdam (KSLA) has two main goals:

1. to establish the characteristic properties that an effective reuse repository for a department in the Laboratory should possess, and

2. to define a process by which such a repository can be created.

These goals are not, of course, unique (cf. [5, 8]). However, they imply some important and not so obvious side conditions we have not encountered in other research efforts. These side conditions will be discussed later on. First, please bear with us while we make some remarks on the goals in order to provide some insight into what we mean by them.

By the characteristic properties of an effective reuse repository we do not only mean technical properties such as the component classification mechanism, tools for component retrieval and the kind of version management, but also more "managerial" ones such as the quality control process and incentive schemes. Characteristic properties should therefore be taken in the widest possible sense.

There is a surprisingly large number of issues that needs to be addressed before any repository can be constructed (see [1, 4]). We will give two examples. First, it is generally agreed that a repository containing only code components has limited benefits ([1]); design or even requirement reuse

is expected to have a higher pay-off potential. There is a practical question-mark, however: source code has a natural representation, but what (formal) language is used to represent designs? There are many possible choices, but little is known about their effectiveness in transporting knowledge from the author of a design to a possible reuser.

The other example is the issue of white-box reuse versus black-box reuse: in the case of black-box reuse the reuser cannot change a code component, whereas in the white-box case he or she has the possibility to change it. The advantages of black-box reuse are that (1) the documentation does not need to be written or rewritten (it is already there), (2) the understanding of the program during maintenance or bug fixing is facilitated and (3) more support can be given in the event of an error being found in a component. The advantages of white-box reuse are that the likelihood that a component can be (partially) reused increases and that partial reuse is more effective than writing from scratch. Although many advocate white-box reuse, in this instance, too, it is not yet clear what the most cost-effective alternative is or, more generally, what the essential factors that are determine which alternative to take.

The second goal takes the reuse technology a step further than the first. The intention of the second goal is to make the construction of an effective repository a clearly defined and manageable task. Moreover, it should make it a *repeatable* effort, *not* relying on reuse researchers. A great deal of effort has already been put in this topic, see, for example, [5, 8].

For the second goal, totally different, but nevertheless related, issues than for the first goal need to be addressed. For example: a characteristic property will be the way in which domain knowledge is represented. For the second goal it will be necessary to describe the process of acquiring such knowledge. Another example concerns the components. For the first goal it is sufficient to know that a component can be a piece of source code or a piece of design (or ...). A question that needs to be answered for the second goal is how components are constructed. Are they developed from scratch or are they to be found in the existing software (salvaging)? If the latter is the case, the question arises as to what kind of tools, techniques, and methods are needed. Furthermore, little is yet known about which alternative is in the long run economically the most attractive: development for reuse or software salvaging.

In the remainder of this paper we will discuss some important properties of software development at the Laboratory, our approach to fulfill the goals as stated in the foregoing, and we will conclude with some remarks.

The Laboratory Environment

An important issue that is related to our environment is that the Laboratory is *not* a software production environment. Software is not an end in itself. At the Laboratory, software is mainly written to *explore* a problem and/or solution. The primary aim of the researcher is generally to gain insight into a physical process. The software is only a means to express and test ideas. Hence much software has a very short lifetime, ranging from a few hours to a few days. Regularly, though, the software developed during research is used for a long time, especially software that has resulted at the end of a research effort. In such a case the software is a vehicle that transports knowledge from the Laboratory to other parts of the company.

Although it might seem that, in such an environment, there is little opportunity for reuse, it has nevertheless been observed that many pieces of functionality are implemented time and again.

In our efforts we hope to find ways to locate and capture these pieces.

The software is written by researchers who are experts in their fields, say, physics and chemistry, but who have received little or no training in software development. This has severe repercussions on several aspects of the repository. First of all, we have to limit ourselves in the use of formal languages. We cannot use a formal language, such as Z or predicate calculus, to specify the source text components. Such a formalism would not be understood easily and it would hinder possible reuse severely. Similarly, we cannot use a very formal language to describe designs and properties of data structures and a transformational approach (see e.g.[2]) to software reuse is also not a viable option. In the end it means that we cannot use formal languages the researcher is not familiar with, unless these are supported to the extent that the formal aspects become hardly noticeable.

The fact that experts in the field are generally not experts in software development influences the domain analysis particularly. Experts in the field eloquently explain the technical difficulties they have solved, but are much less able to explain how their solutions are implemented. For example, they could describe a Fortran subroutine by stating that it is needed by another one that computes, say, mixing in a reactor. Moreover, one could say that their translation of the concepts in the domain to objects in the software is often not up to modern software development standards. For example, the concept of abstract data types is unknown to many.

It goes perhaps without saying, that in the Laboratory the researchers do not follow standard, or even common, software development methods. This implies that a great deal of attention must be given to let researchers reuse software. We expect, unless great care is taken, that the "not-invented-here" syndrome will be a major problem. The "Programmers's Viewpoint" as presented in [10] maps rather too well on researchers in the Laboratory.

Our approach

Apart from the well-known motivations for reusing software, productivity and quality, another one also plays a role. The average employee works in a research job for about five years and then moves on to another, most likely non-research, job. This implies that there is a high staff turnover. It is believed that the results of the domain analysis could be very useful in bringing new employees up to date with the activities of a department.

We have elected to study the construction of a repository for a department in the Laboratory rather than one for the whole Laboratory. The areas in which research is performed are quite diverse. To develop a single repository for the whole Laboratory was therefore, in our opinion, unrealistic. We do expect, however, that repositories for different departments will overlap, and we hope to devise mechanisms that will enable possible reusers to look over the boundaries of the repositories without becoming lost in the sheer quantity of components.

The repository will not only contain source code but designs and code templates as well. We also intend to provide links to the software that was used to test the various components. The 3Cs model (see [3]) will be used to define the structure of a reusable component.

A case study was started one of whose goals was to construct a repository for a department. It was decided to start this case study with a domain analysis. A priori it was decided that the "deliverables"of the domain analysis consists of the following three items:

1. a glossary of terms

In this glossary the concepts and objects of the domain will be defined and their interrelationships recorded. It is expected that many terms in the glossary will have a counterpart in software. It is particularly important to have a clear view of the interrelationships between the terms, because these interrelationships indicate the likely interactions between the software counterparts. As such they (partially) determine the interface of the software components. The glossary also provides important information for the classification of the components.

For reasons explained above, we will use natural language to document the components. To avoid some of the ambiguities that arise in natural language documentation, the author of the documentation is allowed to use only the technical terms that are listed in the glossary with the same meaning. So the glossary will become a constant "terms of reference" or context for the reader of the documentation. This will facilitate the ease of understanding.

2. a list of characteristic problems
   We hope to find components in the large amount of available software or, to be precise, we hope to adapt pieces of existing software such that they become components. The list of characteristic problems is a first step in gaining a grip on the available software and forms a first classification mechanism for it.

3. one or more solution strategies for every characteristic problem
   An example of a characteristic problem is a phase equilibrium computation. In such a computation there is a mixture of chemical substances, a pressure and a temperature. A phase equilibrium computation entails the computation of the number of phases and their compositions. Such a problem can be solved in many different ways: using different models, different numerical methods, etc. The purpose of this "deliverable" is to have a relatively abstract description of each solution. The solution strategies will enable us to further classify the existing software and enable us to abstract away from any specific details of one implementation. This should help us to construct truly reusable components.

Although the domain was not large, already early in the domain analysis phase it was observed that it would require a substantial effort. That the Laboratory is not a software production environment and the researchers not professional software developers hindered the analysis substantially. Since a full domain analysis would take probably too long, we are considering to follow the approach as given in [5].

Notwithstanding the above observation, the domain knowledge representation is much more an open problem than the domain analysis. We are still actively looking for a good "tool." Due to the technical environment a great deal of technical notation exists. We need to have a way to be able to capture such notation, since it would facilitate the understanding of components a great deal. Therefore, we consider a hypertext-like environment in which it is possible to define cards using TEX a viable option. Furthermore, this hypertext-like environment should provide support for layered semantic networks. Such networks will be used to record the "deliverables" of the domain analysis.

Concluding Remarks

We have described some of the problems we are facing in constructing a repository for a department in our Laboratory. The nature of our environment forces us to use natural languages to describe

and/or specify components. We see the domain analysis as the important step in the construction of an effective reuse repository. The process to arrive at a repository will be an adaptation of the process described in [5]. The process for domain analysis is still being developed; our starting point was an approach presented in [6].

We have just started a case study and we have a long way to go before software reuse is institutionalised software reuse.

References

[1] Biggerstaff, T.J., Richter, C. *Reusability Framework, Assessment, and Directions.* IEEE Software, Vol. 4, Nr. 2, March, 1987.

[2] Boyle, J.M. *Program Reusability through Program Transformation.* IEEE Transactions on Software Engineering, Vol. SE-10, Nr. 5, September, 1984.

[3] Frakes, B., Latour, L., Wheeler, T. *Descriptive and Prescriptive Aspects of the 3Cs Model – SETA Working Group Summary* – In: Proceedings of the Third Annual Workshop: Methods & Tools for Reuse, CASE Center, Syracuse University, June, 1990.

[4] Oddy, G. *Software Reuse at G-MRC.* In: [7], pp. 30 - 35.

[5] Prieto-Diaz, R. *Making Software Reuse Work: An Implementation Model.* In: [7], pp. 86 - 92.

[6] Prieto-Diaz, R. *Domain Analysis for Reusability.* Proceedings of COMPSAC'87, 1987, pp. 23 - 29.

[7] Prieto-Diaz, R., Schäfer, W., Cramer, J., and Wolf, S. (eds.) Proceedings of the First International Workshop on Software Reusability, Dortmund, Germany, July 3-5, 1991. SWT-Memo Nr. 57, University Dortmund, Germany.

[8] Tracz, W. *RMISE Workshop on Software Reuse Meeting Summary.* In: [9], pp. 41 - 53.

[9] Tracz, W. *Software Reuse: Emerging Technology.* The Computer Society of the IEEE, 1988, ISBN 0-8186-4846-3.

[10] Tracz, W. *Software Reuse: Motivators and Inhibitors.* Proceedings of COMPCON S'87, 1987, pp. 358 - 363.