Analysing the Return of Investment of Reuse
Giancarlo Succi, Francesco Baruchelli
DISA, Università di Trento,
Via Inama 5, I-38100 Trento, Italia

ABSTRACT
Software reuse can be very useful in increasing the productivity and the quality level of a company, but the costs of this kind of investment are not always covered by the advantages it brings. It is particularly important to be able to correctly analyze the cost implied by software reuse in order to decide whether to adopt it or not. Even when a reuse policy has been adopted, reuse is not always the right solution and there are situations in which it is more convenient to develop a component from scratch. It is difficult to obtain a mathematical model for the evaluation of the effort required by the adaptation of a component in a new context. A different approach is proposed where the estimation is made using the knowledge and the experience gained from previous cases and stored in an ad-hoc base.

INTRODUCTION
When a company decides to adopt a reuse policy in its software development process, there are many factors that must be considered. In fact, reuse is a long term investment that can be very useful if the related techniques are applied correctly, but on the contrary, can be very dangerous and expensive if it is not well organized and planned.

A complete analysis of all the pros and cons of introducing software reuse in a company is not an easy task. In fact, apart from the analysis of the investment needed to build a reusable software library complete and flexible enough to satisfy many of the current and future needs of the company’s engineers, even the more simple decision regarding the opportunity to reuse a component found in the library is often a difficult one. A support to this kind of decision process is the subject of this paper.

In the next sections we will consider the problems of the costs of software reuse, the elements needed for a good “reuse or not” decision process, and the problems related to any mathematical model which tries to represent this particular view of software reuse. Then a new approach based upon Case-Based Reasoning, a technique typical of Artificial Intelligence, is proposed and described, together with a schematic description of its possible integration with a classical reusable software library.

THE OPPORTUNITY OF SOFTWARE REUSE
The reuse of software is often seen as the silver bullet needed to solve all the problems related with the ongoing software crisis. In fact, the application of a reuse policy can be very rewarding in many contests, and gives a better payoff when involving not only the source code, but the products of the earlier phases of the software process, too, i.e. the requirements, the high level architecture and design, and so on ([6]). On the other sides, when adopting software reuse in a development process, one must consider the fact that reuse is not for free ([4]) for a series of reasons. First of all, there is the need of a mid term investment in order to build a library of software entities¹, then, even after this resource has been established the reuse of existing material implies a cost, since an entity can be reused as is and without some changes only in very few lucky situations. This means that the advantages of reusing an entity instead of rebuilding it from scratch should be considered very carefully and compared against the costs that the needed changes require: the decision of whether or not to reuse an existing entity should not have yes as the default answer, but should be taken after carefully considering a series of factors.

The key problem driving the decision process is the following: which is the best solution in terms of the effort needed to obtain the same result. Here the words same result mean that the characteristics of the entity built adapting an existing object to the new context are at least equivalent to the characteristics obtained developing a new component from scratch (this is not an equivalence relation, since the reused entity could have some characteristics better than the ones needed in the new context, e.g. we could reuse a higher quality entity in a context where quality is not a concern and therefore we would not bother to guarantee the same quality level when developing the same entity from scratch).

The data necessary in taking the decision are the following:

• the effort needed for developing the entity from scratch;
• the effort due to the changes needed to adapt the existing entity selected for reuse to the new context.

If we suppose that an entity suitable to be reused has been previously found through the use of an ad hoc artifact library, the previous data can be derived from the

¹ Here we intend with software entity the product deriving from any phase of the software development process, not limiting the definition to raw source code.
description of the needed entity and of the found existing one. The problem can be divided into the following sub-problems corresponding to the classes of data defined above:

- estimate the effort needed for the development of a new entity basing on its description;
- estimate the effort needed for the reuse of an existing entity basing on the description of both the involved entities: the existing and the desired one.

The first problem has been addressed in many works, such as Cocomo and Cocomo2 ([4]) and is outside the scope of this paper, which focuses on the second issue.

THE COSTS OF SOFTWARE REUSE

As we have seen in the previous section, software reuse implies a series of costs which do not arise when developing software in a traditional way. In the following we take into consideration the costs related to the single instances of reuse, i.e. the effort due to the adaptation of the reused entity, ignoring the costs for the introduction of a reuse policy in a company (i.e. the installation and population of a software artifacts library, the adoption of a different development process, and all the issues related to the company as a whole).

The first reason for changing a potential reusable entity is the change of its semantics, e.g. if we have a structure implementing a stack of integers and we need a stack of strings the semantics of the entity has to be changed in order to modify its functionality. Even if this is probably the most natural and immediate source of changes, it is not the only one. In fact, we could have a component with exactly the same semantics of an existing one, but it can not necessarily be reused as is, since some modifications, even major ones, could be needed to make it conform to the quality required in the context of the new project in which it will be used. The same arguments can be applied to the easiness of maintenance and modification of the components and to many other characteristics.

This multiplicity of changes requires that a complex set of issues are considered when evaluating the effort needed for the adaptation of the entity to the new context, and that we do not limit our attention only to the semantic distance between the components. If we try to express the effort needed by the adaptation of the reused entity with a mathematical formula, we must consider each of the factors that can cause the change and their values for the desired and for the existing entity as a parameter in our relation. After all the parameters have been identified, a function relating them to the adaptation effort has to be found. This requires the analysis of a great number of past reuse cases and can lead to results that are not universally valid, since the effort can be affected by the environmental variables of the context in which reuse is applied, such as the adopted programming paradigm (e.g. object-orientation vs. procedural programming), or the process model (e.g. iterative vs. fall). Furthermore, the relations found analyzing the data will probably be not linear but rather complex and hard to represent with a simple mathematical formula. Finally, another great limit of this approach is the fact that it leads to a static model of a reality which is instead very dynamic, since the adoption of new techniques in the development process can cause the model to become rapidly obsolete.

All the mentioned problems can suggest a completely different strategy to address the problem of the adaptation effort estimation, in order to be able to configure in some way the model to adapt it to different realities and let it grow and tune itself through its continuous use. The adaptation and the learning problems are characteristic of the artificial intelligence. If we consider the fact that our knowledge of the part of the world we want to represent is limited to a series of examples, two of the AI branches seems particularly suited to our task:

- neural networks;
- case-based reasoning (CBR).

In the following we will analyze the feasibility of the adoption of CBR to our problem, since a CBR tool is more similar to a classical software library than a neural network, and CBR has effectively been used as a software library ([3]).

CASE-BASED REASONING

CBR is a problem solving method which solves new problems by adapting solutions that were used to solve old problems ([12]). This definition allows us to define the typical components of a case:

- the problem;
- the solution;
- the effect of the solution.

The form in which the related information are represented is very variable and depends upon the different tools and models; for instance Kolodner adopted in CYRUS the dynamic memory model ([7]), while in PROTON the case memory is embedded in a network structure ([10]). Aamodt

---

2 The examples refer to source code for sake of simplicity, but the same issues are valid for other products of the software development process, such as the requirements or the high level architecture of a system, too.
and Plaza ([11]) identified a cyclic process for the solution of new problems, and divided it into the following four phases (depicted in Figure 1):

1. RETRIEVE the most similar case or cases;
2. REUSE the found information and knowledge to attempt to solve the problem;
3. REVISE the proposed solution if necessary;
4. RETAIN the parts of the new solution which are likely to be useful for future problems as a part of a new case.

- the limited previous knowledge of the domain must not be ignored, but can be exploited through the integration of the knowledge base with the CBR.

**THE CBR APPROACH FOR EVALUATING THE REUSE EFFORT**

The first and most important task when trying to apply a CBR approach to the solution of a problem is the correct definition of the problem. This is rather independent from the particular method that will be used and it means that we have to identify the elements that are fundamental in the description of an instance of the problem and of its solution, i.e. the elements which characterize the single cases. Only after this preliminary phase the rules used to infer the distance between different cases and to find the most similar existing solution for a new problem can be discussed.

When trying to evaluate the effort required for the adaptation of a software component to a new context, the solution of the problem is easily identified as the required effort, which can be expressed in many ways, for instance in terms of man-months (or fractions of man-months). The most difficult part in this case is to identify the factors that are relevant in determining the effort. The first of the elements that have to be considered is obviously the semantic distance between the desired and the existing component, as it is calculated by many software artifact libraries basing upon their internal classification mechanisms (e.g. the faceted classification), but as we have seen in the previous sections, this is not enough in order to obtain a sound value for the estimation.

We have identified the following parameters as forming a good descriptor for the cases:

- semantic distance: this has already been discussed;
- available documentation: a well documented component is far more understandable, and therefore, easier to modify, than a poorly documented one;
- dimension of the found component: it appears evident that more a component is big, more it is hard to correctly modify it, since a bigger component is more difficult to understand that a smaller one;
- complexity of the found component: simple components are obviously easier to understand than complex ones;
- number of relations with other components: there can be some interdependencies between different components that can require some extra effort;
- level of the component in the life cycle, i.e. requirement, architecture, code, and so on: the effort required to modify the code must be different than the effort required to modify a requirement even though

![Diagram of CBR phases](image)

**Figure 1: CBR phases.**

Some general knowledge of the domain, for instance a limited set of rules, supports the process, especially during the adaptation phase. We do not go deeper in the analysis of all the different CBR methods and tools which can be found in the literature, but only stress these facts:

- this approach allows an adaptation to different environments, due to the extensive nature of the knowledge. In fact, the cases that are in the Case-Base can be limited to the ones that took place in the same environmental conditions;
- the knowledge is not static, but there is a continuous learning process through the acquisition of new examples;
the semantic distance in the two cases should be the same;

- quality level of the components: this factor is important both in absolute and in relative terms. In fact, in absolute terms, if we need to develop a high quality component the required effort will be higher than the one required for the development of a lower quality component; for the relative part, the effort required when the found component is of higher or equal quality than the needed component will be smaller than the one required if the quality level of the found component is lower than the level we need;

- number of reported bugs: the higher this number, the more the tailoring of the component have to be performed with particular care, with a following rise in the effort;

- version number: this is important to take into account the number of changes already undergone by the component, which can have lowered its understandability;

- compliance to the standards: an extra effort can be required if the component has to be modified not only to fit the new requirements, but to comply to the standards, too;

- creation date of the found component: this can be important if some of the standards and techniques of the company have changed after the creation of the component;

- author of the component: the availability of the author know-how can be very helpful to shorten and improve the tailoring process. Furthermore, the reusability of the components developed by an engineer can be usually higher than reusability of the components developed by another person, and this can emerge from the examples stored in the case-base.

The weights and the proper use of the described factors can be established through the use of a knowledge base composed by a few very simple rules. A full set of rules have not yet been fully identified, but some simple facts can be stated and used in the CBR approach:

- the semantic distance is generally the most important factor in determining the effort required by the changes;

- the availability of the author and of a good documentation for the component is more important when dealing with the code or detailed architecture than when dealing with a requirement that can be considered more self-explaining;

- the quality level is more important for the components produced in the higher phases of the life cycle, since it affects all the following artifacts.

Just like the parameters used to describe the various cases, these and other yet-to-be-discovered rules have must then be coded in a formal language that is strongly dependent from the particular selected CBR method and tool.

INTEGRATION WITH A TRADITIONAL SOFTWARE ARTIFACT LIBRARY

In order to experiment the described approach to the estimation of the reuse effort, an existing libraries of reusable software entities is being modified to be integrated with a CBR shell ([2]) that will be configured to solve our kind of problems.

The reusable software library is Sarto ([13]), a tool featuring the traditional classification methods used for software components ([5]). The most important of these methods for our aims is the faceted classification ([11]), since it allows not only a perfect match, but gives an estimation of the semantic distance between the desired and the found component, too. The other parameters used to describe the various cases that were identified in the previous section are not usually present in the descriptor of a component in a traditional reusable software library, and their specification constitutes an overhead which can be obstacle to the successful application of the method. For this reason we have choose to exploit one of the main characteristics of Sarto: the capability to establish relationships between the different entities stored in the repository. In this way the features which are common for all the components developed in the same project (e.g. the quality level), can be stored only once in a special descriptor representing the project as a whole and then linked through a relationship to all the single entities which share them.

The relationships between the entities can be useful in evaluating the degree of documentation for a component, too. In fact, this can be extracted through some simple rule from the number of entities of type document linked to the found component by a relationship of kind documented by and their quality level.

All the information collected by the library during the search phase can be used to determine the parameters related to the found component that, together with the description of the desired entity, will allow the CBR tool to identify the most similar case in the case base and to estimate the required adaptation effort refining the found solution through some rules of its simple knowledge base. After the end of the integration work, the CBR tool will be simply a part of Sarto completely invisible to the final user. The results of the queries will comprehend the estimation
of the effort required to adapt the found component to the new context.

CONCLUSIONS AND FUTURE WORK

The adoption of a reuse policy requires an investment due to the need of new tools to support all the reuse related activities and the need to develop a software library which should complete enough to guarantee a good percentage of successful reuse. Nonetheless, this investment can be cost effective only if all the activities related to the reuse process are well planned and organized.

Even when the usefulness of software reuse has been recognized by a company and the proper techniques and infrastructures has been introduced, software reuse is not always a solution preferable to the development of a new component completely from scratch, since the adaptation costs can be sensible. For this reason it is very important to correctly evaluate the effort required by the changes for the tailoring of an existing component to its new context. This kind of estimation is a difficult task that can not be modeled efficiently through a simple mathematical model, since too many factors affect the effort which is strongly dependent from many environmental parameters.

We have proposed a different approach which tries to estimate the costs related to the reuse of an existing component through the analysis of past similar cases using the techniques proper of Case-Based Reasoning. The first step has been the identification of the factors that affect the estimation of the reuse effort and of a small set of simple rules that can be used in the search of similar cases and in the adaptation of the found solution to the new problem. This approach to the problem is currently being implemented through the integration of an existing reusable software library with a CBR tool. This requires essentially the addition of some new modules and functionalities to the library, and the encoding of the identified factors and rules in the formal language characteristic of the CBR tool. In order to prove the effectiveness of the method, it will be very important to fill not only the library with components, but the case base with examples from the real world, too. The proposed approach seems promising essentially for its flexibility and suitability to different environments thanks to its continuous learning process through the acquisition of new cases. This allows a continuous refinement and improvement of the accuracy of the estimation, too. A possible drawbacks is the initial step needed for the start-up, since it requires the presence of a sufficient number of cases from the past and can not be very useful when this kind of experience lacks, and the system must rely only on its simple knowledge base.

REFERENCES


J.Kolodner, Retrieving Events From Case Memory. *Proceedings from the Case-Based Reasoning Workshop*, DARPA, Clearwater Beach, 1988, pp. 233-249.


APPLIED COMPUTING REVIEW
A Publication of the ACM Special Interest Group on Applied Computing
Volume 4 Number 1 Spring 1996

Table of Contents

Special Issue on Parallel Object-Oriented Programming

Guest Editors: Boleslaw K. Szymanski, Charles D. Norton

M. Austern, R. Towle, A. Stepanov:
Range Partition Adaptors: A Mechanism for Parallelizing STL

S. Baden:
Software Infrastructure for Non-Uniform Scientific Computations on Parallel Processors

D. Gannon, S. Diwan, E. Johnson:
HPC++ and the Europa Call Reification Model

Y. Ishikawa:
MPC++ Approach to Parallel Computing Environment

A. Chien:
ICC++ - A C++ Dialect for High Performance Parallel Computing

C. Kesselman:
High Performance Parallel and Distributed Computation in Compositional CC++

C. Norton, V. DeCyk, B. Szymanski:
On Parallel Object Oriented Programming in Fortran 90