Chapter XII
Decision Rule for Investment in Reusable Code

Roy Gelbard
Bar-Ilan University, Israel

ABSTRACT

Reusable code helps to decrease code errors, code units and therefore development time. It serves to improve quality and productivity frameworks in software development. The question is not HOW to make the code reusable, but WHICH amount of software components would be most beneficial (i.e. cost-effective in terms of reuse), and WHAT method should be used to decide whether to make a component reusable or not. If we had unlimited time and resources, we could write any code unit in a reusable way. In other words, its reusability would be 100%. However, in real life, resources and time are limited. Given these constraints, decisions regarding reusability are not always straightforward. The current chapter focuses on decision-making rules for investing in reusable code. It attempts to determine the parameters, which should be taken into account in decisions relating to degrees of reusability. Two new models are presented for decisions-making relating to reusability: (i) a restricted model, and (ii) a non-restricted model. Decisions made by using these models are then analyzed and discussed.

INTRODUCTION

Software reuse helps decrease code errors, code units and, therefore, development time; thus improving quality and productivity of software development. Reuse is based on the premise that educing a solution from the statement of a problem involves more effort (labor, computation, etc.) than inducing a solution from a similar problem for which such efforts have already been expended. Therefore, software reuse challenges are structural, organizational and managerial, as well as technical.

Economic considerations and cost-benefit analyses in general must be at the center of any discussion of software reuse; hence, the cost-benefit issue is not HOW to make the code reusable, but

WHICH amount of software components would be most beneficial (i.e. cost-effective for reuse), and WHAT method should be used when deciding whether to make a component reusable or not.

If we had unlimited time and resources, we could write any code unit in a reusable way. In other words, its reusability would be 100% (reusability refers to the degree to which a code unit can be reused). However, in real life, resources and time are limited. Given these constraints, reusability decisions are not always straightforward.

Literature review shows that there are a variety of models used for calculating-evaluating reuse effectiveness, but they are not focused on the issue of the degree to which a code is reusable. Thus the real question is how to make reusability pragmatic and efficient, i.e. a decision rule for investment in reusable code. The current chapter focuses on the parameters, which should be taken into account when making reusability degree decisions. Two new models are presented here for reusability decision-making:

- A Non-Restricted Model, which does not take into account time, resources or investment restrictions.
- A Restricted Model, which takes the aforementioned restrictions into account.

The models are compared, using the same data, to test whether they lead to the same conclusions or whether a contingency approach is preferable.

BACKGROUND

Notwithstanding differences between reuse approaches, it is useful to think of software reuse in terms of attempts to minimize the average cost of a reuse occurrence (Mili et al 1995).

[Search* + *(*1-p*) * (ApproxSearch* +q ** Adaptation old + ***(*1-q*)* Development*** new ***)]***

Where:

- ***Search*** (***ApproxSearch***) is the average cost of formulating a search statement of a library of reusable components and either finding one that matches the requirements exactly (appreciatively), or being convinced that none exists.
- ***Adaptation*** old is the average cost of adapting a component returned by approximate retrieval.
- ***Development*** new is the average cost of developing a component that has no match, exact or approximate, in the library.

For reuse to be cost-effective, the aforementioned must be smaller than:

p ****Development* exact **+*(*1-p*)* q * Development*** approx **+*(*1-p*)* (*1-q*)´ Development*** new***)***

Where:

- ***Development*** exact and ***Development*** new represent the average cost of developing custom-tailored versions of components in the library that could be used as is, or adapted, respectively. Note that all these averages are time averages, and not averages of individual components, i.e. a reusable component is counted as many times as it is used.

Developing reusable software aims at maximizing P (probability of finding an exact match) and Q (probability of finding an approximate match), i.e. maximizing the coverage of the application domain, and minimizing adaptation for a set of common mismatches, i.e. packaging components, in such a way that the most common old mismatches are handled easily. Increasing P and Q does not necessarily mean putting more components in the library; it could also mean adding components that are more frequently