Chapter X
Decision Rule for Investment in Frameworks of Reuse

Roy Gelbard
Bar-Ilan University, Israel

ABSTRACT

Reuse helps to decrease development time, code errors, and code units. Therefore, 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, that is, 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 are limited and there are clear deadlines to be met. Given these constraints, decisions regarding reusability are not always straightforward. The current research focuses on decision-making rules for investing in reuse frameworks. 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 decision-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

Reuse helps decrease development time, code errors, and code units, thereby improving quality and productivity frameworks in 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, 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, that is, 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 are limited and there are clear deadlines to be met. Given these constraints, reusability decisions are not always straightforward.

A review of the relevant literature shows that there are a variety of models used for calculating-evaluating reuse effectiveness, but none apparently focus on the issue of the degree to which a code is reusable. Thus, the real question is how to make reusability pragmatic and efficient, that is, a decision rule for investment in reuse frameworks. The current study 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 above-mentioned 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 research in terms of attempts to minimize the average cost of a reuse occurrence (Mili, Mili, & Mili, 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 above 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, that is, 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), that is, maximizing the coverage of the application domain and minimizing adaptation for a set of common mismatches, that is, 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 needed, because adding components not only has its direct expenses (adaptation costs), but also increases search costs.

There are two main approaches to **code adaptation**: (1) identifying components that are generally useful and (2) covering the same set of needs with fewer components, which involves two paradigms: (i) abstraction and (ii) composition. Composition supports the creation of a virtually unlimited number of aggregates from the same set of components, and reduces the risk of combinatorial explosion that would result from enumerating all the possible configurations. In general, the higher the level of abstraction at which composition takes place, the wider the range of systems (and behaviours) that can be obtained. The combination of abstraction and composition provides a powerful paradigm for constructing systems from reusable components (Mili et al., 1995).

Frakes and Terry describe a wide range of metrics and adaptation models for software reuse. Six types of metrics and models are reviewed: cost-benefit models, maturity assessment models, amount of reuse metrics, failure modes models, reusability assessment models, and reuse library metrics (Frakes & Terry, 1996).

Other studies (Henninger, 1999; Otso, 1995; Virtanen, 2000; Ye, Fischer, & Reeves, 2000; Ye & Fischer, 2002), present additional metrics and methods, evaluate and make comparisons, but as is typical in an emerging discipline such as systematic software reuse, many of these metrics and models still lack formal validation. Despite this, they are used and are found useful in industrial practice (Ferri et al., 1997; Chaki, Clarke, Groce, Jha, & Veith, 2004).

Empirical work (Mens & Tourwé, 2004; Paulson, Succi, & Eberlein, 2004; Tomer, Goldin, Kuflik, Kimchi, & Schach, 2004; Virtanen, 2001; Ye, 2002) has analyzed existing reuse metrics and their industrial applicability. These metrics are then applied to a collection of public domain software products and projects categories to assess the level of correlation between them and other well-known software metrics such as complexity, volume, lines of code, and so forth.

Current research is focused on decision-making rules for investment in reuse frameworks. The well-known "simple model" and "development cost model" deal with these decisions, but do not take into account restrictions and constraints such as time, budget, resources, or other kinds of investment, such as delivery time, that may impact on the decision to reuse.

ANALYZING NEW REUSE MODELS

Assume a software development project contains 3 code components: A, B, and C, and we need to determine two things: Which of these components should be reusable? What criteria should be taken into account?

There are eight combinations—choice alternatives for these 3 components, as shown in Table 1 (+ represents "make reusable," - represents "don't make reusable").

A. The Non-Restricted Model

The model contains the following parameters:

- **Ci**—cost of creating component **i** from scratch (without making it reusable).
- **Ri**—cost of making component **i** reusable (extra costs – not included in **Ci**).
- **ICi**—cost of implementing reusable component **i** into code.
- **NRi**—number of reuses of component i.

(C, R, and NR are in man-hours).

Savings resulting from making component **i** reusable are represented as follows:

$$SAVi = NRi * (Ci - ICi) - (Ci + Ri)$$

Therefore: If **SAVi** > 0, it is worthwhile to make component **i** reusable.

Suppose a company that employs two kinds of programmers: **M** and **N**. Programmers of type **M**

Table 1. Choice alternatives

Alternative	Component A	Component B	Component C
1	-	-	-
2	+	-	-
3	-	+	-
4	+	+	-
5	-	-	+
6	+	-	+
7	-	+	+
8	+	+	+

are permanent employees of the firm. Programmers of type **N** are highly qualified consultants who are employed by the company for specific projects. The company is going to write/create/develop a new project, and has to make a decision regarding which components should be reusable.

The following are additional parameters:

- **Cim**—hours needed for programmer M to create component **i** from scratch.
- **Rim**—hours needed for programmer M to make component **i** reusable.
- **ICim**—hours needed for programmer M to implement reusable component **i** into code.
- **Sm**—costs of programmer M, per 1 hour.

Hence:

Ci= Min(Cim*Sm, Cin*Sn)
Ici= Min(ICim*Sm, ICin*Sn)
Ri= Min(Rim*Sm, Rin*Sn)

Hence:

SAVi = NRi *(Min(Cim*Sm, Cin*Sn) – Min(ICim*Sm, ICin*Sn)) – (Min(Cim*Sm, Cin*Sn) + Min(Rim*Sm, Rin*Sn))

B. The Restricted Model

The non-restricted model has the following limitations:

- It requires absolute values
- It is quite difficult to measure parameters such as: **Ci**, **Ri,** and **Ici**
- It does not take into account the most typical situation where time and budget are restricted as well as in-house investment in reuse, that is, time and resources for reusable code developing.

In order to avoid these limitations, the restricted model is based upon the following parameters:

- **I**—maximal **investment** that can be allocated for writing a reusable code.
- **T**—maximal calendar **time** that can be allocated for writing a reusable code.
- **Ii**—percent of "**I"** needed to make component **i** reusable.
- **Ti**—percent of "**T"** needed to make component **i** reusable.
- **Ci**—relative **complexity** of creating component **i** from scratch.
- **Fi—frequency** (%) of future projects that are likely to reuse component **i.**
- **Pi**—relative **profit** of making component **i** reusable.
- $\mathbf{R_I}$**—r**emainder of "**I"**, after some reusable components have been written.
- $\mathbf{R_T}$**—**remainder of "**T**," after some reusable components have been written.

Assume that: **Pi** = **Ci * Fi.**

Hence: component **i** is the next component to be made reusable if:

Pi = Max(P1, P2, ..., Pn-1, Pn)
Ii <= **RI**
Ti <= RT

C. Illustrative Example: Non-Restricted Model

The following example (Example 1) demonstrates the decision made by the non-restricted model. Assume we want to develop 10 projects, each one containing components A, B, and C according to Table 2.

Hence**:** **NRa** = 10, **NRb** = 1, **NRc** = 4

Table 3 presents illustrative assumptions concerning **Cim** and **Cin** (hours needed for programmer type M and N to create component **i** from scratch).

Moreover, assume programmers' costs to be: **Sm** = 20, **Sn** = 40

Hence:

Ca = Min(300*20, 200*40) = 6,000
Cb = Min(20*20, 10*40) = 400
Cc = Min(150*20, 100*40) = 3,000

Table 4 presents illustrative assumptions concerning **Rim** and **Rin** (hours needed for programmers type M and N to make component **i** reusable).

Hence:

Ra = Min(650*20, 300*40) = 12,000
Rb = Min(15*20, 7*40) = 280
Rc = Min(150*20, 80*40) = 3,000

Table 5 presents illustrative assumptions concerning **ICim** and **ICin** (hours needed for programmers type M /N to implement reusable component **i** into code).

Hence:

ICa = Min(60*20, 15*40) = 600
ICb = Min(5*20, 3*40) = 100
ICc = Min(50*20, 10*40) = 400

Table 2. Example 1, number of components for future reuse

Project	1	2	3	4	5	6	7	8	9	10
Component A	+	+	+	+	+	+	+	+	+	+
Component B	+									
Component C	+	+	+	+						

Table 3. Example 1, Ci illustrative assumptions

Programmer type	Component A	Component B	Component C
Type M	*300*	*20*	*150*
Type N	*200*	*10*	*100*

Table 4. Example 1, Ri illustrative assumptions

Programmer type	Component A	Component B	Component C
Type M	*650*	*15*	*150*
Type N	*300*	*7*	*80*

Table 5. Example 1, ICi illustrative assumptions

Programmer type	Component A	Component B	Component C
Type M	*60*	*5*	*50*
Type N	*15*	*3*	*10*

Hence:

SAVa = 10 *(6,000 – 600) – (6,000 + 12,000) = 36000 > 0
SAVb = 1 *(400 – 100) – (400 + 280) = -380 < 0
SAVc = 4 *(3000– 400) – (3,000+ 3,000) = 4400 > 0

In light of the mentioned, the reuse decision according to the non-restricted model is to make components A and C reusable (i.e., alternative 6).

D. Illustrative Example: Restricted Model

The following example (Example 2) demonstrates the decision made by the restricted model, based on the previous example (Example 1). Assume the following:

1. **I**—10,000.
2. **T**—150. The available remaining time to make the existing code reusable.
3. **Ci**—assume component B is the easiest one to develop, and requires 10 hours. Assume component A requires 300 hours and component C requires 150 hours. Hence, complexities are: C_A=30, C_B=1, C_C =15.
4. **Fi**—component A will be reused by 100% of future projects, B by 10%, and C by 40%.
5. $\mathbf{I_A}$ = 12,000/10,000 = 120%, $\mathbf{I_B}$ = 280/10,000 = 2.8%, $\mathbf{I_C}$ = 3000/10,000 = 30%.
6. $\mathbf{T_A}$ = 300/150 = 200%, $\mathbf{T_B}$ = 7/150 = 4.7%, $\mathbf{T_C}$ = 150/150 = 100%.

Hence Example 2 parameters are shown in Table 6.

Taking time and investment restrictions into account, the reuse decision, according to the restricted model is to make only component C reusable (i.e., alternative 5).

CONCLUSION AND FUTURE TRENDS

The current study presented two new reuse decision making models: a restricted model and a non restricted model, which mainly differ in the way they take into account real-life constraints-restrictions such as time, budget, and resources repetition.

The models produced different results from the same data. The decision made by the restricted model pinpointed fewer software components for reuse. It is worth mentioning that different groups of software components were not the issue, but rather different subgroups of the same group, that is, software components selected by the restricted model were subgroups of components selected by the non-restricted model.

Table 6. Parameters used by Example 2

Component	Ci	Fi(%)	Pi	Ii(%)	Ti(100%)
A	*30*	*100*	*30*	*120*	*200*
B	*1*	*10*	*0.1*	*2.8*	*4.7*
C	*15*	*40*	*0.6*	*30*	*100*

Moreover, the parameters of the restricted model relate to relative value arguments, by contrast to the parameters of non-restricted model, which relate to absolute values. While absolute values are difficult to measure, relative values are simpler to define. There are a variety of formal methods by which relative values may be defined, methods that are used in other areas of software engineering, such as cost estimation, effort estimation, priority decision, and others.

The reusability decision made by the restricted model may be biased by the following parameters: time, resources, component complexity, and number-percent of future projects in which the component would be reused. Further research should be conducted, focusing on decision robustness in light of the mentioned parameters and their possible spectrum.

ACKNOWLEDGMENT

I would like to thank Tami Shapiro for her contribution to this work.

REFERENCES

Chaki, S., Clarke, E. M., Groce, A., Jha, S., & Veith, H. (2004). Modular verification of software components. *IEEE Transactions on Software Engineering, 30*(6), 388-402.

Desouza, K. C., Awazu, Y., & Tiwana, A. (2006). Four dynamics for bringing use back into software reuse. *Communications of the ACM, 49*(1), 96-100.

Ferri, R. N., Pratiwadi, R. N., Rivera, L. M., Shakir, M., Snyder, J. J., Thomas, D. W. et al. (1997). Software reuse: Metrics for an industrial project. In *Proceedings of the 4th International Symposium of Software Metrics* (pp.165-173).

Fischer, G., & Ye, Y. (2001). Personalizing delivered information in a software reuse environment. In *Proceedings of the 8th International Conference on User Modeling* (p. 178).

Frakes, W., & Terry, C. (1996). Software reuse: Metrics and models. *ACM Computing Surveys, 28*(2), 415-435.

Henninger, S. (1999). An evolutionary approach to constructing effective software reuse repositories. *ACM Transactions on Software Engineering and Methodology, 6*(2), 111-140.

Kirk, D., Roper, M., & Wood, M. (2006). Identifying and addressing problems in object-oriented framework reuse. *Empirical Software Engineering, 12*(3), 243-274.

Mens, T., & Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering, 30*(2), 126-139.

Mili, H., Mili, F., & Mili, A. (1995). Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering, 21*(6), 528–562.

Otso, K. J. (1995). *A systematic process for reusable software component selection* (Tech. Rep.). University of Maryland.

Paulson, J. W., Succi, G., & Eberlein, A. (2004). An Empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering, 30*(4), 246-256.

Reifer, D. J. (1997). *Practical software reuse*. Wiley.

Spinellis, D. (2007). Cracking software reuse. *IEEE Software, 24*(1), 12-13.

Tomer, A., Goldin, L., Kuflik, T., Kimchi, E., & Schach, S. R. (2004). Evaluating software reuse alternatives: A model and its application to an industrial case study. *IEEE Transactions on Software Engineering, 30*(9), 601-612.

Virtanen, P. (2000). Component reuse metrics—Assessing human aspects. In *Proceedings of the ESCOM-SCOPE* (pp. 171-179).

Virtanen, P. (2001). Empirical study evaluating component reuse metrics. In *Proceedings of the ESCOM* (pp. 125-136).

William, B., Frakes, W. B. & Kang, K. (2005). Software reuse research: Status and future. *IEEE Transactions on Software Engineering, 31*(7), 529-536.

Ye, Y. (2002). An empirical user study of an active reuse repository system. In *Proceedings of the 7th International Conference on Software Reuse* (pp. 281-292).

Ye, Y., & Fischer, G. (2002). Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the International Conference on Software Engineering* (pp. 513-523).

Ye, Y., Fischer, G., & Reeves, B. (2000). Integrating active information delivery and reuse repository systems. In *Proceedings of ACM-SIGSOFT 8th International Symposium on Foundations of Software Engineering* (pp. 60-68).

KEY TERMS

Decision Rule: Either a formal or heuristic rule used to determine the final outcome of the decision problem.

Non-Restricted Reuse-Costing Model is a reuse-costing model that does not take into account real-life constraints and restrictions, such as time, budget, resources, or any other kind of investment.

Reuse: Using an item more than once. This includes conventional reuse where the item is used again for the same function and new-life reuse where it is used for a new function (wikipedia).

Reuse-Costing Model: A formal model, which takes into account the expenditure to produce a reusable software product.

Restricted Reuse-Costing Model: A reuse-costing model that takes into account real-life constraints and restrictions such as time, budget, resources, or any other kind of investment.

Software Reuse: Also called code reuse, is the use of existing software components (e.g., routines, functions, classes, objects, up to the entire module) to build new software.