Determining the Value of a Corporate Reuse Program

Jeffrey S. Poulin
Joseph M. Caruso

Systems Software Development Tools
International Business Machines Corporation

Abstract

Reuse metrics must accurately reflect the amount of effort saved. We must have realistic measures to ensure the credibility of the value we place on reuse and to separate reuse benefits from those of other technologies also competing for limited investment dollars. This paper defines a reuse metrics and Return On Investment (ROI) model at IBM that distinguishes reuse savings and benefits from those already gained through accepted software engineering techniques. Used in conjunction with a planned reuse program, the potential of reuse serves as a powerful motivator. We derive three reuse metrics from readily available software data elements and use these metrics in a return on investment model that establishes a sound business justification for reuse.[1]

1.0 Overview

Management traditionally uses software metrics to quantify the software development process. With an emerging technology, however, metrics must extend beyond their traditional role; applying the reuse metrics must also encourage the practice of reuse. Most organizations do not practice formal reuse or remain reluctant to invest in a formal reuse program. Reuse metrics must assist in the technology insertion process by providing favorable process improvement statistics and by placing emphasis on activity conducive to reuse. Our experience shows great motivation results from establishing a positive financial return using a sound ROI model.

When reviewing reuse experience reports, the reader must consider the results in their reported contexts. The interpretation of any measurement depends on the collection method, environment, and numerous other factors. Most results lack sufficient supporting information. The reader therefore does not have the details required to repeat the experiment or case study, understand the results, or cogently agree with or disagree with the conclusions [19]. This paper explains the method IBM uses to quantify and report reuse.

1.1 Using versus Reusing Code

Many organizational reuse programs consist of simply recovering old code for later use (sometimes referred to as “unplanned reuse”). However, the greatest benefits from reuse result from engaging in a formal, planned reuse program. *When* an organization makes the reuse decision distinguishes between these two classes of reuse [20]. Planned reuse starts early in the software lifecycle; a thorough requirements study and domain analysis of the problem area provide evidence of planning for reuse. This additional planning and domain analysis identifies factors that normally change in later projects, such as:

1. Hardware or System Software,
2. User, Mission, or Installation, and,
3. Function or Performance.

Early design and analysis will result in components that can accommodate these changes without modification. The organization pays for the increased quality level in planned reuse in the initial development of the component. However, the organization recovers this cost quickly when subsequent projects reuse the generalized software and by reduced support costs resulting from having only one base product to maintain. Unfortunately, most organizations fail to plan for reuse during their software development except for the occasional use of existing software in the new application. Although many programmers practice this *ad hoc* reuse, traditional software development methods do not normally include the systematic reuse of existing code. We must distin-

[1] *Proceedings of the IEEE Computer Society International Software Metrics Symposium,* Baltimore, MD, 21-22 May 1993, pp. 16-27.

guish between these two classes when determining the financial return of a reuse program because of the much greater value planned reuse has to the organization.

Table 1 contains a summary of how system changes relate to the classes of reuse.

Table 1. Relation between system changes and classes of reuse.

Required Change	**Code Recovery Only**	**With Planned Reuse**
Hardware or System Software	Rehosting	Porting
User, Mission, or Installation	Retargeting	Tailoring
Function or Performance	Salvaging	Assembling

1.1.1 How to measure code recovery and planned reuse

Planned reuse provides the greatest cost and productivity benefits compared to unplanned reuse because planned reuse results in having only one base product to maintain. Positive financial benefits result most quickly from planned reuse. Therefore, reuse metrics should focus on planned reuse. Although code recovery has benefits, we typically only experience these benefits in the development phase of the lifecycle. Furthermore, we cannot generally determine the value of reuse when modifying code; variables include how much code required change and the amount of effort expended to understand the code prior to making the changes. Some organizations choose to track the amount of recovered code in their products to emphasize the amount of "total leverage" they gain by copying and modifying old software. However, we do not include code recovery in these reuse metrics.

Most organizations practice reuse by assembling reusable components into new applications; metrics must capture this activity. Some more advanced organizations tailor software through parameters. Tailorable software results from thorough domain analysis and careful program design. The tremendously successful reuse experiences on the IBM Advanced Automation System (ASS) for the Federal Aeronautics Administration [14] provide an example of tailoring; reuse metrics must also capture this activity.

The third form of planned reuse results from porting. However, porting can cause misleading economic projections in reuse metrics because we normally include porting in the the business planning of products. We make resource estimates for products based on development of the product on one hardware platform or operating system. We then allocate a relatively nominal amount of resources for changes required to adapt to other environments. Although reuse includes porting, reuse metrics should not claim savings due to porting because we have established porting as part of the business and development planning process.

A further reason to exclude porting from reuse metrics comes from the nature of the activity. Porting normally involves adapting a minor portion of a large product. To include ported code in reuse metrics would cause misleading results in the form of unrealistically high measures of reuse activity. For example, an organization making small changes to a large base might report levels of "reuse" close to 100%,[2] whereas an organization performing an equal amount of labor on an original project might do very well to to demonstrate reuse levels of 5-10%. When used in an ROI model, these levels of reuse result in unrealistic financial savings. To prevent this distortion, IBM does not include code porting in these reuse metrics. Organizations that port software separately track the amount of porting required to deliver their products.

Table 2 lists how we measure the various classes of code recovery and planned reuse.

Table 2. Reuse Techniques and Reuse Metrics

Code Recovery	**Measured?**		**Planned Reuse**	**Measured?**
Rehosting	No		Porting	Separately
Retargeting	No		Tailoring	Yes
Salvaging	No		Assembling	Yes

1.2 Measuring Reuse

[2] This exceeds the theoretical maximum of 85% [13].

1.2.1 When to measure reuse

We have differentiated between code recovery, which results in new software to maintain, and planned reuse, in which we assemble or tailor products from building blocks of reusable software. Next, we define reuse based on "who" uses the component.

Central to improving the practice of reuse is the understanding that we expect good design and management within development organizations, and although many groups work closely together, we find communication less common between organizations. Communication, essential for the simple exchange of information and critical to sharing of software, becomes more difficult as the number of people involved grows and natural organizational boundaries emerge. Therefore, measurements must encourage reuse *across* these organizational boundaries.

We determine that for a component to count as *reuse*, an organization that did not develop and does not maintain the component must use the component. Because software development organizations vary we define a "typical" organization as either a programming team, department, or functional group of about eight people or more. Also, although organizational size may indicate how well communication within and between organizations takes place, we must also consider functional boundaries between organizations. For example, a small programming team may qualify as an organization if it works independently.

For consistency, we report the type and size of the reporting organization when reporting reuse metrics. This provides an informal check on the flexibility allowed in selecting the most appropriate boundary for the organization. Selection of an inappropriate small boundary would distort the value of the metrics upward and an inappropriate large boundary would result in low reuse values. Changing the organizational boundary between reports would eliminate any possibility for comparisons and evaluation of the reuse program.

1.2.2 Reusing versus Using Components

Many organizations report the reuse percent of a product or a new release of a product. The organizations intend to indicate the portion of their product (normally expressed in lines of code, or LOC) saved by reusing software from other products or product releases. In either case, the effort attributed to reuse comes from completely unmodified reusable components. We make reusable components easy to identify in new products by using straightforward criteria; if use of a component saved having to develop a similar component, we record it as reuse. However, although an organization may "use" a component numerous times, an organization can "reuse" a component only once.

We make this distinction to increase the accuracy of our return on investment analysis of projects. Since we expect organizations to develop subprograms for often-needed services and to use components previously developed for a product or previously developed by themselves, we do not credit them with savings resulting from these activities. We consider this good software development, a practice expected from every programmer.

In one actual example, a project reported 11 kloc of reuse on a relatively small application. Closer inspection revealed 5120 lines of the 11 kloc came from one 10-line reusable macro and that all 5120 lines came from the same module. A code review revealed that the original code:

```
Do i := 1 to 512
        MACRO(i);
```

consisted of 2 instructions (the DO..WHILE and the call to MACRO) and 10 reused instructions. However, the optimizer automatically unrolled the loop to yield:

```
MACRO(1);
MACRO(2);
  ....
MACRO(511);
MACRO(512);
```

The report therefore contained 512 source instructions and 5120 reused instructions. This does not accuratly reflect the productivity or reuse on the project. For this reason, IBM considers that an organization may "use" a part numerous times, but an organization may "reuse" a part only once.

1.2.3 Units of Measurement

These metrics use traditional "lines of code" to quantify the effort in software development. Although lines of code (LOC) have well known deficiencies as a unit of measure [12], people understand them, people collect them, and people compare them. Nonetheless, we may take actions to increase confidence when using LOC as a measure. A code counting tool may standardize the way we count LOC. Another action, taken with the metrics in this paper, involves using metrics derived from ratios or percentages of effort. This eliminates the units of "LOC" from the metrics. Since LOC make a good indicator of overall effort [21], then the portion of LOC reused makes a good indicator of effort saved.

2.0 Reuse Metrics

2.1 Observable Data

We calculate reuse metrics from the following observable data elements which IBM [11] and other companies [6] have used for many years. Managing the software process depends on complete and accurate data; sensible reuse metrics require certain data about the process. We can usually measure observable data directly from the product. For example, we can directly measure the different classes of source instructions. Observable data may also come from historical files that we collected for a variety of reasons related to managing the software development process. Historical data that IBM normally collects include costs for software development, defect rates, and maintenance costs.

Shipped Source Instructions (SSI). The total lines of code in the product source files.

New and Changed Source Instructions (CSI). The total lines of code new or changed in a new release of a product.

Reused Source Instructions (RSI). The total lines not written but included in the source files. RSI includes only completely unmodified reused software components.

Source Instructions Reused By Others (SIRBO). The total lines of code that other products reuse from a product.

Software Development Cost. A historical average required for estimating reuse cost avoidance.

Software Development Error Rate. A historical average required for estimating maintenance cost avoidance.

Software Error Repair Cost. A historical average required for estimating maintenance cost avoidance.

When acquiring the observable data elements, especially RSI, we must recognize when reuse actually saves effort. This requires the analyst to distinguish reuse from normal software engineering practices (e.g., structured programming) and to eliminate implementation-dependent options effecting the observable data. (e.g., static versus dynamic sub-program expansion). For example, the programmer's decision to implement a system service as a subroutine, which gets expanded at runtime, or as a macro, which gets expanded at compile time, should not affect the reuse metric. Therefore, the LOC measurements must come from the unexpanded source files of the product.

2.1.1 Shipped Source Instructions (SSI)

Shipped Source Instructions (SSI) indicate the number of non-comment instructions in the source files of a product. SSI do <u>not</u> include Reused Source Instructions (RSI). A call to a reusable part counts as one SSI. When reporting reuse measures for development organizations, SSI include all the source instructions the organization maintains. The lines of code actually written by someone for a product contribute to the SSI.

2.1.2 Changed Source Instructions (CSI)

Changed Source Instructions (CSI) indicate the number of non-comment source instructions that programmers wrote <u>new, added, or modified</u> in a product release. CSI does <u>not</u> include Reused Source Instructions (RSI) or unchanged base instructions from prior releases of the product. CSI includes all source instructions from partially modified components. A call to a reusable part counts as one CSI.

2.1.3 Reused Source Instructions (RSI)

Reused Source Instructions (RSI) indicate the source instructions an organization ships but does not develop or maintain. RSI come from completely unmodified components. Programmers normally locate these components in a reuse library. Base instructions from prior releases of a product and source instructions from partially modified parts do not count as RSI.

Source Instructions from a reused part count once per organization independent of how many times one calls or expands the part.

2.1.4 Source Instructions Reused by Others (SIRBO)

Source Instructions Reused by Others (SIRBO) indicate the amount of an organization's source instructions that other organizations reuse. This indicates how much an organization contributes to reuse. A successful reuse program depends not only on organizations reusing software but also them helping other organizations reuse software. SIRBO measures both the parts contributed for use by others and the success of those parts. Organizations writing successful reusable parts will have a very high SIRBO because SIRBO increases every time another organization reuses their software. This encourages organizations to generate high quality, well-documented, and widely-applicable reusable components.

To measure SIRBO, for each part an organization contributes to a library, take the sum of:

$$\begin{gathered} \mathit{SIRBO} = (\mathit{Source\ instructions\ per\ part}) \\ \times (\mathit{The\ number\ of\ organizations\ using\ the\ part}) \end{gathered}$$

Example: If an organization contributes the following to a reuse library: a 10kloc module in use by 5 other departments, a 25kloc macro in use by 6 other departments, and an unused 75kloc macro, the organization's SIRBO equals:

$$\begin{gathered} \mathit{SIRBO} = (5\ \mathit{depts.} \times 10\ \mathit{kloc}) + (6\ \mathit{depts.} \times 25\ \mathit{kloc}) \\ + (0\ \mathit{depts.} \times 75\ \mathit{kloc}) = 200\mathit{kloc} \end{gathered}$$

Like RSI, the number of times the same organization invokes or calls the part does not affect SIRBO. The same rules apply for counting RSI; use of a reusable part saved having to develop the part one time, not one time for every call to the part. SIRBO changes with time. As more organizations reuse the components, the SIRBO of the donating organizations increases.

2.1.5 Software Development Cost (Cost per LOC)

To determine the financial benefit of reuse, we must know the cost of developing software without reuse. We generally can obtain this new software development cost from historical data kept by the financial planners and management of the organization. If necessary, calculate the new software cost by adding all the expenses of the organization, including overhead, and dividing by the total output (in LOC) of the organization.

2.1.6 Software Development Error Rate (TVUA rate)

No amount of testing, inspection, or verification can guarantee that we have released a product without errors. Although emphasis on quality and strict adherence to development processes leads to better products, errors inevitably reveal themselves in the marketplace. Every development organization has a historical average number of errors, or *TVUAs* (“Total Valid Unique Program Analysis Reports”) uncovered in their products.

Note that we normally design and test software components built for reuse to stricter standards than those for normal program product components. We justify the additional cost of this effort by the savings gained when other organizations do not have to develop and maintain a similar component. The additional testing results in increasing the quality of the component, so we can expect fewer errors from reusable code.

2.1.7 Software Error Repair Cost (Cost per TVUA)

To quantify the benefit of the increased quality of reusable components, we need the historical average cost of maintaining components with traditional development methods. As with software development cost, we generally obtain this figure from financial planners and management in the organization. If necessary, calculate the software error repair cost by taking the sum of all costs, including overhead, of repairing latent errors in software maintained by the organization and dividing by the number of errors repaired. Although software maintenance includes enhancements to products, the cost of increasing function does not factor into the software error repair cost.

2.2 Derived Metrics

The observable data elements combine to form three derived reuse metrics. The first two metrics indicate the level of reuse activity in an organization as a percentage of products and by financial benefit. The third metric includes recognition for writing reusable code [17].

1. Reuse Percent; the primary indicator of the amount of reuse in a product or practiced in an organization. Derive Reuse Percent from SSI, CSI, and RSI.
2. Reuse Cost Avoidance; an indicator of reduced total product costs as a result of reuse in the product. Derive Reuse Cost Avoidance from from SSI, CSI,

RSI, TVUA rates, software development cost (cost per LOC), and maintenance costs (Cost per TVUA).

3. Reuse Value Added; an indicator of leverage provided by practicing reuse and contributing to the reuse practiced by others. Derive Reuse Value Added from SSI, RSI, and SIRBO.

2.2.1 Reuse Percent

The Reuse Percent indicates the portion of a product, product release, or organizational effort that we attribute to reuse. The importance of the Reuse Percent comes from the ease of calculation and ease of understanding. Unfortunately, it has little value without a supporting framework. Many companies report their reuse experiences in terms of "reuse percent" but few describe how they calculate the values. They commonly include unplanned reuse in the value, making it difficult to assess actual savings or productivity gains. Since we clearly define RSI, we believe our Reuse Percent metric reasonably reflects the development effort saved.

Reuse Percent of a product

To calculate the Reuse Percent of a product (or first release of a product):

$$ReusePercent = \frac{RSI}{RSI + SSI} \times 100\%$$

Product Example: If a product consists of 65kloc SSI and an additional 35kloc from a reuse library, then find the Reuse Percent of the product by:

$$ReusePercent = \frac{35\ kloc}{35\ kloc + 65\ kloc} \times 100\% = 35\%$$

Reuse Percent of a product release

For a new release of a product, RSI comes from reusable components that did not appear in previous releases (or the product base) of the product. We count calls to component used in previous releases as one new or changed source instruction (CSI). Calculate the Reuse Percent of a product release using:

$$ReusePercent = \frac{RSI}{RSI + CSI} \times 100\%$$

Product Release Example: If a release of a product consists of 7k CSI plus 3k "new" RSI from a reuse library, then calculate the Reuse Percent for this product release as:

$$ReusePercent = \frac{3\ kloc}{3\ kloc + 7\ kloc} \times 100\% = 30\%$$

Reuse Percent for an organization

All software developed and maintained by an organization counts as the organization's SSI. Any software used by the organization but maintained elsewhere counts as their RSI. We calculate the Reuse Percent of an organization using:

$$ReusePercent = \frac{RSI}{RSI + SSI} \times 100\%$$

Organizational Example: If a programming team develops and maintains 80k SSI and the team additionally uses 20k RSI from a reuse library, then the Reuse Percent for the team is:

$$ReusePercent = \frac{20\ kloc}{20\ kloc + 80\ kloc} \times 100\% = 20\%$$

2.2.2 Reuse Cost Avoidance (RCA)

The Reuse Cost Avoidance estimates the financial benefit of reuse. When used with a clear definition of RSI, this metric shows the tremendous return on investment potential of reuse. Because we use RCA when performing return on investment (ROI) analysis of reuse programs, RCA helps with the insertion of reuse technology.

Reusing software requires less resources than new development, but does not come free. The developer still must search for, retrieve, and assess the suitability of reusable components before finally choosing the appropriate part for integration into the product. Although reuse requires effort to understand and integrate reusable parts, studies show that the cost of this effort consists of only about 20% of the cost of new development [22], [7]. Based on this *relative cost of reuse*, we estimate the financial benefit attributable to reuse during the development phase of a project:

$$\begin{gathered}\textit{Development Cost Avoidance}\\ = RSI \times (1 - .2) \times (\textit{New Code Cost})\end{gathered}$$

However, we typically spend only about 40% of the software life cycle in development [10]; significant maintenance benefit also results from reusing quality software. We may estimate this benefit as the cost avoidance of not fixing errors (TVUAs) in newly developed code [8].

We can represent this savings as:

$$\begin{aligned}&\textit{Service Cost Avoidance}\\ &= RSI \times (\textit{TVUA Rate}) \times (\textit{Cost per TVUA})\end{aligned}$$

The total Reuse Cost Avoidance becomes:

$$\begin{aligned}&\textit{Reuse Cost Avoidance}\\ &= \textit{Development Cost Avoidance}\\ &+ \textit{Service Cost Avoidance}\end{aligned}$$

Example: If an organization has a historical new code development cost of \$200 per line, a TVUA rate of 1.5/kloc, and a cost to fix a TVUA of \$43k, then we calculate the RCA for integrating 20k RSI into a product by:

$$\begin{aligned}&\textit{Reuse Cost Avoidance}\\ &= (20\ \textit{kloc} \times .8 \times \$200\ \textit{per line})\\ &+ (20\ \textit{kloc} \times 1.5\ \textit{TVUA per kloc} \times \$43k\ \textit{per TVUA})\\ &= \$3.2\ \textit{million} + \$1.29\ \textit{million}\\ &= \$4.49\ \textit{million}\end{aligned}$$

2.2.3 Reuse Value Added (RVA)

The previous two metrics measure how much organizations reuse software. We must also motivate contributions of reusable software. The Reuse Value Added provides a a metric that reflects positively on organizations that both reuse software and help other organizations by developing reusable code.

We use a ratio, or productivity index for the RVA; organizations with no involvement in reuse have an $RVA = 1$. Contributions to reuse and reusing software positively correlate with the ratio. An $RVA = 2$ indicates the organization becomes approximately twice as effective as the same organization without reuse. In this case, the organization doubled its productivity (or effectiveness) either directly (by reusing software) or indirectly (by maintaining software that other organizations use). Observe that we do not take into account the additional costs of writing code for reuse nor the relative costs of reusing code. We seek only an indicator of the multiplicative effect of planned reuse in an organization. To represent the total effectiveness of a development group, use:

$$\textit{Reuse Value Added} = \frac{(SSI + RSI) + SIRBO}{SSI}$$

Example: A programming team maintains 80kloc and uses 20 kloc from a reuse library. In addition, five other departments reuse a 10kloc module the programming team contributed to the organizational reuse library. Calculate the RVA of the programming team as:

$$\begin{aligned}&\textit{Reuse Value Added}\\ &= \frac{(80\ \textit{kloc} + 20\ \textit{kloc}) + (5\ \textit{depts.} \times 10\ \textit{kloc})}{80\ \textit{kloc}} = 1.9\end{aligned}$$

In this example, the RVA of 1.9 indicates the programming team became approximately 1.9 times more effective than the same team without reuse.

Some organizations organize to obtain the most benefit possible from reuse. For example, the Mid-Hudson Valley Programming Laboratory and the IBM Federal Systems Company in Rockville, Maryland dedicate programming teams to develop and maintain shared software or site-wide reuse libraries. Corporate parts centers, such as the Boeblingen software center, also develop and maintain software for IBM-wide use. Experience shows that although these types of teams may have modest values for the Reuse Percent metric, they have extremely high values for the RVA metric. This high RVA indicates the tremendous programming leverage they provide to their organizations.

3.0 Reuse Return on Investment

The reuse metrics quantify, encourage, and standardize counting methods for projects and development organizations. However, we require a more detailed accounting of costs when developing reuse business cases. Furthermore, we must make a distinction between project level business cases and the business case for the entire corporation. The corporate business case includes overhead costs which we do not attribute to any individual project. We find both types of business cases effective for promoting reuse throughout the corporation.

3.1 Project Level ROI

We find many product managers reluctant to invest in a comprehensive reuse program since the benefits of writing reusable code will often accrue to projects outside their realm of responsibility. Therefore, any definition of return on investment should include benefits which other projects reap as a result of efforts by the initiating project. A straightforward formulation for return on investment follows:

$$ROI = RCA + ORCA - ADC$$

Where:

- *ROI* represents the Return on Investment that would occur in infinite time
- *RCA* represents the Reuse Cost Avoidance for the initiating project
- *ORCA* represents the Reuse Cost Avoidance for other projects benefiting from the reusable code written by the initiating project
- *ADC* represents the Additional Development Cost of writing reusable code to the initiating project

To keep business case analysis for small projects simple, the above formula ignores the time value of money. We can justify this because most small projects have a limited duration. For ORCA, we previously defined RCA as the sum of Development Cost Avoidance and Service Cost Avoidance to the initiating project. We calculate ORCA similar to RCA but we base ORCA on SIRBO rather than RSI. To calculate ORCA, sum the RCA for each benefiting project:

$$ORCA = \sum_{i=1}^{n} SIRBO_i \times (1 - \textit{relative cost of reuse}_i) \times (\textit{New Code Cost}_i) + \sum_{i=1}^{n} SIRBO_i \times \textit{TVUA Rate}_i \times \textit{Cost per TVUA}_i$$

Where:

- $SIRBO_i$ represents the Source instructions reused by project i
- n represents the number of projects reusing code written by the initiating project
- $\textit{relative cost of reuse}_i$ represents the cost of integrating reusable code for project i relative to the cost of creating a new line of code, which we take as 1
- $\textit{New Code Cost}_i$ represents the cost per line of code for project i
- $\textit{TVUA Rate}_i$ represents the number of TVUAs per kloc for project i
- $\textit{Cost per TVUA}_i$ represents the cost to repair an TVUA for project i

We define Additional Development Cost (ADC) as:

$$(\textit{relative cost of writing reuse} - 1) \times \textit{Code written for reuse by others} \times \textit{New code cost}$$

Where:

- *relative cost of writing reuse* represents the cost of writing reusable code relative to the cost of writing code for one time use, which we take as 1. Usually we set this number to 1.5 [7].
- *Code written for reuse by others* represents the kloc of code written for reuse by the initiating project

Example: Assume an organization has a Reuse Cost Avoidance of \$4.49 million as in the previous example. Additionally the organization has developed 15 kloc of reusable code for other projects with a relative cost of writing reuse of 1.5. Two projects (Project A and B) have already agreed to reuse some components and have the following data:

Project	**SIRBO**	**Cost/ LOC**	**Rel Cost**	**TVUA rate**	**Cost/ TVUA**
A	2	200	.2	2	10
B	8	80	.3	.5	18

Then:

$$\begin{aligned}
&ROI = RCA + ORCA - ADC \\
&= \$4.49\ \textit{million} + ORCA - 15 \times (1.5 - 1) \times \$200\ \textit{per line} \\
&= \$4.49\ \textit{million} + ORCA - \$1.5\ \textit{million} \\
&= \$2.99\ \textit{million} \\
&+ (2\ \textit{kloc} \times .8 \times \$200\ \textit{per line}) \\
&+ (2\ \textit{kloc} \times 2\ \textit{TVUA/kloc} \times \$10K\ \textit{per TVUA}) \\
&+ (8\ \textit{kloc} \times .7 \times \$80\ \textit{per line}) \\
&+ (8\ \textit{kloc} \times .5\ \textit{TVUA/kloc} \times \$18K\ \textit{per TVUA}) \\
&= \$3.87\ \textit{million}
\end{aligned}$$

Although straightforward, the number of computations can make the final figure for ROI prone to error. For this reason we wrote a menu-driven program which provides a business template for project level ROI analysis. The program presents an input menu to the user, complete with defaults for many of the input parameters previously described. The user alters parameters as required and enters the unique project level data. In addition to providing the ROI for the project reuse program, the program displays all three reuse metrics.

The automation of the reuse metrics and ROI computations greatly assists the software project manager in developing justification to implement a successful reuse program at the project level.

3.2 Corporate Level ROI

A corporate level reuse program can consist of many project-level reuse programs. The benefits that accrue to the corporation consist of the sum of the benefits to the individual projects (their RCA). From a cost perspective, however, we must also consider reuse program start-up activities. For instance, we might establish a group of people to promote reuse programs across the corporation. We might develop tools to store, search for, and retrieve reusable parts. We may have to purchase hardware to process or to store the reusable parts. We may have to purchase parts from outside vendors rather than develop them locally. The corporate ROI analysis must include these costs.

Additionally, the start-up activities for a corporation can require a significant period of time before the reuse infrastructure begins to yield productivity and quality savings. For this reason, a corporate level ROI should take into account the time value of money. We express this ROI through the internal rate of return (IRR) approach where we select k so that:

$$C_0 = \frac{R_1 - C_1}{1 + k} + \frac{R_2 - C_2}{(1 + k)^2} + \ldots + \frac{R_n - C_n}{(1 + k)^n}$$

Where:

C_0 represents the corporate reuse start-up costs
R_i represents the revenue (savings) in year i
C_i represents the costs in year i
n represents the number of years to consider for revenues
k represents the discount rate

If we let k_0 equal the corporation's cost of capital then we can define the related net present value (NPV) metric which measures the value of the corporate reuse program in dollars:

$$NPV = -C_0 + \frac{R_1 - C_1}{1 + k_0} + \frac{R_2 - C_2}{(1 + k_0)^2} + \ldots + \frac{R_n - C_n}{(1 + k_0)^n}$$

Figure 1 shows an example of a corporate level ROI. In this example, business planning practices dictated that we consider benefits for 5 years into the future. The hypothetical ROI results in a hefty \$20 million net present value and with 104% internal rate of return. Although this ROI seems extraordinarily high by conventional business standards, it does not reflect the risk inherent in many of the underlying assumptions of the ROI. For instance, we must make assumptions about the growth of reuse over time, the relative cost of reuse, the cost of writing reusable code, the amount of vendor purchased reusable code versus code written internally within the corporation, etc. Since we must base these assumptions on historical averages and other debatable factors, we must examine the effects of varying these assumptions on the ROI.

Benefits/Costs	Year					
	Start	1	2	3	4	5
Benefits						
Total DCA	0	6763	11870	17990	23713	30447
Total SCA	0	1646	877	395	169	71
Total RCA	0	8409	12747	18385	23882	30518
Support Costs						
Reuse Technology Center	85	88	65	41	29	30
Site Champions	241	500	598	717	858	1024
Disk Storage	0	182	321	481	621	779
Total Support	326	770	984	1239	1508	1833
Other Costs						
Total ADC	0	3730	5299	6247	6009	5008
Tool Development	1200	1200	1200	1200	1200	1200
Vendor Parts	2400	1450	1450	1450	1450	1450
Net Savings per year	-3926	1259	3814	8249	13715	21027
Present Value	-3926	1049	2649	4774	6614	8450
Net Present Value	\$19,610K					
Internal Rate of Return	104%					

Figure 1. Example Corporate Level ROI (K\$)

Notice that the sample ROI includes costs for tool development and support costs such as personnel responsible to communicate the benefits of reuse thereby helping to spread reuse throughout the corporation and individual sites. These costs do not incur to any individual project but we must include them in the corporate ROI.

4.0 Related Work

Since an essential step in assessing the success and effectiveness of a reuse program includes quantifying results, several groups have reported on reuse metrics. However, the uniqueness of these metrics lies in the attention given to the definition of RSI and in attempting to present reuse as “real effort saved.” Although [1] differentiates between reuse within an organization and reuse from sources external to the organization, no other paper addresses how to measure the classes of reuse nor do they provide a concentrated definition of RSI.

Related work includes metrics to assess the reusability of components and to determine the suitability of a component for use in a new application [4], [5]. However, our work focuses on measuring the amount of reuse and the value of reuse to an organization. Although we find that many organizations use "reuse percent," the majority of published methods focus on financial analysis. Other organizations also find that the financial benefit of reuse convinces project managers and planners to invest in formal reuse programs [15].

In 1988 Gaffney and Durek [8] published a comprehensive model addressing business case analysis of reuse. They premise their model on the need to amortize the cost of the reuse program, including the additional cost to build reusable components, across all projects using the component. When doing cost benefit analysis for software reuse, one needs to consider the long-term benefits and associated costs which apply to every project using the component. They argue that a better economic estimate includes the number of times the organization reuses the component.

Other groups present cost-benefit models of reuse [16] and [2]. The NATO model consists of listing the major benefits and costs of reuse, then applying time-value of money formulas to adjust for future values. The Barns and Bollinger model determines the cost/benefit of reuse by subtracting investment costs of producing reusable software from the reduced consumer development costs minus adaptation costs.

4.1 Future Work

Future work includes further validation of these measures and ROI model. This includes comparing the predicted versus actual costs avoided and comparing increased productivity rates with the values calculated in the metrics and the ROI model. Although the model uses industry experience for default values in the equations, we use actual values when we have them. Future work will include methods to quantify reuse in areas other than software (e.g., Design, Test Case, Information Development).

4.2 Conclusion

Managing the software process depends on sound business decisions based on accurate measurements. This paper introduces the following three metrics and a return on investment model for software reuse: Reuse Percent, Reuse Cost Avoidance, and Reuse Value Added. The metrics rely on easy to collect data, provide reasonable representations of reuse activity, and encourage reuse. Most importantly, the metrics provide reliable input to the corporate reuse ROI model, where we carefully and realistically define the benefits attributable to reuse.

With emerging technologies such as software reuse, the value of metrics and ROI analysis goes beyond the traditional benefits of assuring the quality of reusable components, demonstrating the success of a program, and improving the ability to plan and predict for future projects. They also serve to encourage reuse by providing feedback on the results of a reuse program and by highlighting the benefits of a organizational reuse effort.

We made the ROI business template program available to all sites to help convince project management that they should implement a formal reuse program. The managers have effectively put the program to use in a number of real cases. The IBM Reuse Technology Support Center uses the corporate level ROI to evaluate the relative benefit of formal reuse to other technologies which improve programmer productivity and quality. Because of the high relative benefit of reuse, the corporate reuse program effectively competes for corporate funds designated for the promotion of new technologies. In addition, several levels of executives have reviewed the analysis and become more aware of the potentially large returns for reuse and the importance of increased focus on formal reuse programs.

4.3 Acknowledgements

We would like to thank the members of the IBM Corporate Reuse Council (CRC) for their contributions to these metrics and supplying data for the ROI analysis. We would also like to acknowledge that portions of this paper also appear in [18].

5.0 Cited References

[1] Banker, Rajiv D., Robert J. Kauffman, Charles Wright, and Dani Zweig, “Automating Output Size and Reusability Metrics in an Object-Based Computer Aided Software Engineering (CASE) Environment,” *Technical Report,* 25 August 1991.

[2] Barns, B.H. and T.B. Bollinger, “Making Reuse Cost Effective,” *IEEE Software,* Vol 8., No.1, Jan, 1991, pp. 13-24.

[3] Bourland, D. David and Paul Dennithorne Johnston, ed., To Be or Not: An E-Prime Anthology, *International Society for General Semantics,* San Francisco, CA, 1991.

[4] Basili, Victor R., H. Dieter Rombach, John Bailey, and Alex Delis, “Ada Reusability Analysis and Measurement,,” *Empirical Foundations of Information and Software Science V,* Atlanta, GA, 19-21 Ocotber 1988, pp. 355-368.

[5] Caldiera, Gianluigi and Victor R. Basili, “Indentifying and Qualifying Reusable Software Components,,” *IEEE Software,* Vol. 24, No. 2, February 1991, pp. 61-70.

[6] Daskalantonakis, M.K., “A Practical View of Software Measurement and Implementation Experiences within Motorola,” *IEEE Transactions on Software Engineering,* Vol. 18, No. 11, November 1992, pp. 998-1010

[7] Favaro, John, “What price reusability? A case study,” *Ada Letters,* Vol. 11, No. 3, Spring 1991, pp. 115-24.

[8] Gaffney, John E., Jr. and Thomas Durek, “Software Reuse- Key to Enhanced Productivity; Some Quantitative Models,” *Software Productivity Consortium, SPC-TR-88-015,* April 1988.

[9] Gaffney, J.E., Jr. and Durek, T.A., “Software Reuse- Key to Enhanced Productivity: Some Quantitative Models,” *Information and Software Technology,* 31:5, June 1989.

[10] “Software Engineering Strategies,” *Strategic Analysis Report,* Gartner Group, Inc., April 30, 1991.

[11] “Corporate Programming Measurements (CPM),” V 4.0, *IBM Internal Document,* 1 November 1991.

[12] Firesmith, D.G., “Managing Ada projects: the people issues,” *Proceedings of TRI Ada '88,* Charleston, WV, USA 24-27 Oct. 1988, pp. 610-19

[13] Jones, T.C. “Reusability in Programming: A Survey of the State of the Art,” *IEEE Transactions on Software Engineering,* Vol. SE-10, No.5, September, 1984.

[14] Margano, Johan, and Lynn Lindsey, “Software Reuse in the Air Traffic Control Advanced Automation System,” paper for the *Joint Symposia and Workshops: Improving the Software Process and Competitive Position,* 29 April-3 May 1991, Alexandria, VA.

[15] Margano, Johan, and Thomas E. Rhoads, “Software Reuse Economics: Cost Benefit Analysis on a Large Scale Ada Project,” *Proceedings of the International Conference on Software Engineering,* 11-15 May 1992, Melbourne, Australia, 1992, pp.338-348.

[16] “Standard for Management of a Reusable Software Component Library,” *NATO Communications and Information Systems Agency,* 18 August 1991.

[17] Poulin, Jeffrey S. and W.E. Hayes, “IBM Reuse Methodology: Measurement Standards,” *IBM Corporation Document Number Z325-0682,* 16 July 1992.

[18] Poulin, Jeffrey S. and Joseph M. Caruso “A Reuse Measurement and Return on Investment Model,” *Proceedings of the Second International Workshop on Software Reusability,* Lucca, Italy, 24-26 March 1993.

[19] Rombach, H. Dieter, “Design Measurement: Some Lessons Learned,” *IEEE Software,* March 1990, pp.17-25.

[20] “Repository Guidelines for the Software Technology for Adaptable, Reliable Systems (STARS) Program,” CDRL Sequence Number 0460, 15 March 1989.

[21] Tausworthe, R.C, “Information models of software productivity: limits on productivity growth,” *Journal of System Software,* Vol 19, No. 2, Oct, 1992, pp. 185-201.

[22] Tracz, Will, “Software Reuse Myths,” *ACM SIGSOFT Software Engineering Notes,* Vol. 13, No. 1, Jan 1988 p. 17-21.

6.0 Biography

Jeffrey S. Poulin joined IBM's Reuse Technology Support Center, Poughkeepsie, New York, in 1991 as an advisory programmer. His responsibilities include developing and applying corporate standards for reusable component classification, certification, and measurements. Dr. Poulin has worked in the area of software reuse since 1985 and helped lead the development and acceptance of the IBM software reuse metrics. He participates on the IBM Corporate Reuse Council, the Association for Computing Machinery, and Vice-Chairs the Mid-Hudson Valley Chapter of the IEEE Computer Society. A Hertz Foundation Fellow, Dr. Poulin earned his Bachelors degree at the United States Military Academy at West Point, New York, and his Masters and Ph.D. degrees at Rensselaer Polytechnic Institute in Troy, New York.

Joseph M. Caruso works in the IBM Systems Programming Tools organization in Poughkeepsie, New York, as an advisory systems analyst. His responsibilities include determining the return on investment for the many technologies funded by his organization. He joined IBM in 1978 as a systems analyst for Material Requirements Planning Systems. In 1985 he moved into Assurance to pursue his interests in statistics. While in assurance, he developed software statistical tools, projected quality levels of software projects, automated reporting and analysis systems, and provided statistical education for the quality assurance organization. His interests include applications of software reliability growth modelling, sample size determination, and Monte Carlo simulation. He received his Bachelors degree from S.U.N.Y. at Stony Brook, his Masters from Penn State University and currently pursues a Doctorate degree from Union College, New York.