Transition Parameters For Successful Reuse Business

Jasmine K.S.
Dept. of MCA, R.V. College of Engineering
Bangalore-59, Karnataka
India

1. Introduction

For any industrial organizations, improving the business performance often means the improvement in their software development performance. The growing popularity of developing the software using reusable components could dramatically reduce development effort, cost and accelerate delivery. Software professionals generally need most help in controlling requirements, coordinating changes, managing (and making) plans, managing interdependencies, and coping with various systems' issues. Since the energy spent on these and similar problems generally consumes a large part of every software professional's time, these are where management can provide the most immediate help.

The primary function of ongoing business management is to monitor the progress of the rollout plans towards achieving these business goals, and to adjust expectations and schedule to meet business and organizational realities. Detailed measurement is crucial to ensure that overall reuse business goals are met. Management should clearly define what is meant by reuse to different parts of the organization, and how to match reuse measurements to business goals of the organization. Reuse measurements must take into account political and organizational realities. When setting up a measurement program, it is must decide what kind of metrics, or what changes in current metrics, are needed to successfully manage a reuse business at the hands-on level. Most of the metrics that software managers use today are aimed at a standalone project group, which does its own estimating, requirements capture, architecting, analysis, design, implementation, and testing.
The need of adjustment in goals and metrics and changes in process and development and implementation methods are mandatory to become success in the transition to reuse business process.

The chapter mainly focuses on the results obtained from my own study and also the lessons learned from the literature survey. In writing this chapter, I have incorporated ideas; suggestions and experience of leading software reuse experts working in various software companies. I am convinced that the lessons and insight provided in this chapter will be of crucial value to any company dependent on software reuse.

2. Transition management –An essential step towards New Business Opportunities

The interest in system innovations is motivated by environmental and economic reasons. The alternative systems should be attractive not only from an environmental point of view but also from an economic point of view (in terms of generating ROI and services to end-users). It is accepted that any system innovations will have disadvantages, which may or may not be overcome. The solution to this problem is the simultaneous exploration of multiple options and adaptive policies, based on iterative and interactive decision making. New systems should not be implemented but "grown" in a gradual manner, relying on feedback and decentralized decision-making (Larry, 2002).

> – *"Radical innovation is the process of introducing something that is new to the organization and that requires the development of completely new routines, usually with modifications in the normative beliefs and value systems of organization members."* -- Nord and Tucker, *Routine and Radical Innovations*, 1987

Transitions here refer to important changes in functional systems. They involve multi-level changes through which an organization fundamentally changes. The transitions are required in the areas such as

- Economy
- Culture
- Management
- Technology

For a transition to occur different development have to come together causing a path of development based on new practices, knowledge, social organization and different guiding principles.

2.1 What exactly is transition management?

Transition management is a new steering concept that relies on 'darwinististic' processes of variation and selection. It makes use of "bottom-up" developments and long-term goals both at the organizational and process level (Garcia, 2003). Learning and institutional change are key elements which means that transition management not so much concerned with specific outcomes but rather with mechanisms for change. The basic philosophy is that or goal-oriented modulation: the utilization of ongoing developments for business and organizational goals. An important question therefore is: what do people really want, both as users and developers?

Collective choices are made "along the way" on the basis of learning experiences at different levels. Different trajectories are explored and flexibility is maintained, which is exactly what a manager would do when faced with great uncertainty and complexity: instead of defining end states for development he sets out in a certain direction and is careful to avoid premature choices.

2.2 Elements of transition management

Transition management consists of following elements:

1. Development of long-term visions
2. Organizing a transition arena for a transition theme
3. Monitoring and evaluation of experiments and transition processes
4. Implementation and Monitoring of transition mechanisms
5. Learning and adjusting to support for future transitions

Key elements of the transition management cycle are: anticipation, learning and adaptation. The starting point is the structuring of problems – to achieve a common outlook. This is followed by the development of long-term visions and goals. Goals are being set via the process and deliberations in transitions arenas. The management acts as a process manager, dealing with issues of collective orientation and adaptation of policy. It also has a responsibility for the undertaking of strategic experiments and programmes for system innovation. Control policies are part of transition management. Transition management aims for generating "momentum" for sustainability transitions. Not all companies will contribute to a transition, but once a new development takes shape, others will follow suit, including companies invested in the old system. When this happens the change process becomes a force of its own. This is a critical phase in a transition in which also unwanted path dependencies occur. Management has to develop assessment tools to measure the effectiveness of this transition process. Transition management requires continuous anticipation and adaptation (Graves, 1989).

2.3 Policy integration

The integration of various policy areas is part of transition management. Areas for integration are: Technological policy, infrastructural policy, adoption & decision making policy and innovation& implementation policy (across organization). This is an important but difficult task. The use of transition agendas and transition arenas should help to achieve this. Policy integration is probably aided by a more open approach of policy making in which learning is institutionalized. Policy renewal is officially part of transition policy. Interactive approach is a better coordination of different policies.

Some findings about adoption policy (Rotmans, 2001):

- Learning and professional development are important to organizations both before and after the adoption
- The adoption decision should be communicated to every developer at all levels
- Adoption should be treated as a continuous process

2.4 What makes transition management different?

Transition management does not rely on blueprints but relies on iterative decision making in which also goals may change. Decisions are made on the basis of experiences and new insights. Policy choices would be more based on long-term desirability instead of on short-term solutions. Long-term possibilities are given support but still need to prove themselves to customer needs. This way, customer may discover what is best.

Through transition management space is being created for change. The space should not be too narrow, lest organization will get locked into suboptimal solutions. To prevent this from

happening, transition management opts for a portfolio approach and 'evolutionary' steering.

2.5 Preparing for Transition management

Transition management is well-accepted in software industry. The management is ready to support it. The transition management builds upon interests and "movements" (change processes) in development methodology. Transition management helps to establish a partnership with business and to stimulate new business based on sustainable innovation.

Transforming a business and its core processes to compete effectively is not just a technology issue however. A successful approach will require careful collaboration between (Griss et al., 1993):

- People e.g. stakeholders, accountability, internal politics)
- Technology e.g. service enablement / integration, governance, business process management/ workflow analytics)
- Processes e.g. improved business / IT alignment, delivery methodology, consolidation/rationalization continuous process improvement initiatives).

Only through careful alignment and management of these three distinct disciplines will an organization embark on a successful business transformation journey.

The recent history of successful market transformations consistently demonstrate that embracing relevant technology principles, standards and industry best practice are all differentiating actors between mere market participation and market leadership. Latest advances in architectural principles such as loosely coupled distributed systems, separation of logic from implementation and common information models all indicate the importance of service orientated techniques in the design and development of flexible and hence sustainable solutions. These new architectural approaches coupled with complementary project delivery methodologies help provide the agility required to liberate existing assets and better align IT with ever changing business needs in timely and repeatable fashion.

2.6 Effects of successful transition

The software industry has just started with transition management. In the short term, few results in terms of reduced development cost, Quality improvement, time-to-market and realization of new business opportunities are to be expected. Expectations are rather high, whereas transitions research shows that transitions defy control and effective steering. Policy can do little more than increase the chance for a transition to occur and shape the features of it. This is also what transition management tries to do by way of evolutionary steering, oriented at processes of variation and selection. Processes of adaptation, learning and anticipation are institutionalized through transition management. Conditions for success and application are: 'sense of urgency', leadership, commitment, willingness to change political culture (based on short-term goals), active management, guidance, trust, and willingness to invest the right resources at right time (Graves, 1989).

It is hoped that the commitment to sustainability transitions helps to make such choices, but whether this will happen is far from certain. Transition management is not an instrument but a framework for policy-making and governance. It is believed that transition management offers an interesting model for policy & governance, combining the advantages of incrementalism (do-able steps which are not immediately disruptive) with those of planning (articulation of desirable futures and use of goals).

2.7 Transition to a Reuse Business

There are a number of approaches to deal with large-scale organization and process change that can be used by a software organization to introduce or improve its reuse practice. Among the approaches mentioned above, by focusing on process and organizational models and dealt with people issues, one can have a successful transition to reuse business.

The proper implementation of following steps will guarantee a successful reuse transition process:

Step1.Assess reuse feasibility by assessing business opportunities and needs, organizational readiness and observing the kind and variety of application systems produced, sort of process, tools and technology used.

Step2. Prepare a transition plan to meat the long term goals (e.g. with emphasis on product line approach)

Step3.Address people issues such as awareness about reuse, convincing reasons for reuse, fear for reuse etc.

Step4.Allow the reuse program to grow naturally and mature through a distinct well defined stages.

Step5. Customize and adapt the traditional software engineering approach to Component system engineering and Application family engineering.

Step 6.Testing, appropriate tool development and deployment of the steps1-5.

3. Assess Reuse Risks and Costs

"*The use of commercial products can have profound and lasting impact on the spiraling cost and effort of building Defense systems, particularly information systems. It is important, however, to remember that simply* '*using COTS*' *is not the end in itself, but only the means.*"--David Carney

Component based software development is becoming more generalized, representing a considerable market for the software industry. The perspective of reduced development costs, shorter life cycles, lower cost of sustainment, and better quality acts as motivation factors for this expansion. However, several technical issues remain unsolved before software component's industry reaches the maturity exhibited by other component industries. Problems such as the component selection by their integrators, the component

catalogs formalization and the uncertain quality of third-party developed components, bring new challenges to the software engineering community. Even the reuse actually slowing it down and making the overall design of the system unwieldy, and unstable. It can actually increase the long-term cost of the system. Requirements, algorithms, functions, business rules, architecture, source code, test cases, input data, and scripts can all be reused. Architecture reuse is the primary means for achieving the cost savings potential of code reuse.
To exploit reuse, the development team must recognize the realities of reused products (Jacobson et.al, 1997).

- The product being reused must closely match the need that it is trying to solve. Otherwise, the components with which it interfaces will become more complex.
- The reused product should be well documented with a well-understood interface in the perspective of future maintenance.
- The reuse product should have been designed for a scenario similar to that for which it is going to be used. If not, extensive testing should be conducted to verify the correctness of the product. Changes in the operational environment may require further overhead in testing
- The reuse product should be stable. Any change to the product must be incorporated into the current development.
- The key to successful reuse is determining the components and functions that are needed before forcing a decision to reuse products.
- The quality of the components and their usage has impact on the software process itself. Introducing software components of unknown quality may have catastrophic results
- The requirements should be made flexible in order to support the usage of components
- The third party certification can be made compulsory to ensure that the components confirm to well-defined standards and are adequate to fulfill the given requirements.

Reuse is a powerful technique, because it allows functionality to be provided rapidly to the user. A decision to reuse a software product is also a constraint on the development team. Rather than being allowed to structure the system in the most effective way possible, a decision to reuse software limits the options available.
Also the reuse of commercially available software has some unique disadvantages. Commercial entities are motivated to limit a user's ability to change products. This is often accomplished by including and then recommending the use of proprietary features. In addition, the update cycle for a commercial package can be a significant drain on maintenance resources. The costs of keeping personnel current in and then integrating new product releases should be assessed as a part of overall life cycle costs.

3.1 Risk Analysis

"Reuse is like a savings account. Before you collect any interest, you have to make a deposit, and the more you put in, the greater the dividend attributed to" -Ted Biggerstaff, 1983 ITT Reuse workshop.

There were two types of approaches I could observe from the survey, for establishing a reuse program: centralized and distributed.

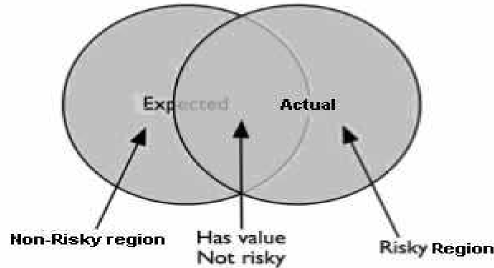

Fig. 1. Prediction region for acceptable values

Fig 1 tells about the prediction region for acceptable values– tend to give good information about value in reuse context. Each region will coincidentally give some information about the other regions, but neither is sufficient by itself. Concentrating on the basis of expectation will not tell as much as information needs to know about risk. The figure gives an insight of savings expected in terms of percentage of reuse used.

	Centralized	Distributed
Risk Factors		
	1) Cost of maintaining a dedicated team (60%)	1) Difficult to coordinate asset development responsibilities (40%)
	2) Cost of training employees (40%)	2) Reluctant to make their own investment for others (60%)
	3) Need for strong management commitment (55%)	3) Need of convincing cost-benefit model (45%)

Table 1. Risk factors identified from the survey

New professional competences in the development teams are required to deal with the introduction of the changes for reuse in the software process. The majority of the estimation models produced during the last decades are found not suitable for reuse products. A discussion on the research difficulties in the software measurement area (which is, of course, crucial to estimation) can be found in (Poulin, 1994). The shift to a new approach to software development requires new estimation models to better capture the essence of CBD. Due to the novelty of CBD and related estimation models, there is still a lack of past experience in which one can support his estimation efforts. An example of a face lifted model that aims

supporting cost estimation in CBSE is the COCOTS (Abts et al.2000), an evolution from COCOMO II(Boehm et al.1995,2000).

"Reuse is something that is far easier to say than to do. Doing it requires both good design and very good documentation. Even when we see good design, which is still infrequently, we won't see the components reused without good documentation". - D. L. Parnas, Software Aging, 16th International Conference Software Engineering, 1994

There are many informal arguments that make software reuse an appealing and economically. In the following section I will discuss some models and theories that have been developed to assess economics of software products and the developed reuse models based on my study. The majority of work on economics of reuse is on the reuse of source code.

3.2 Cost Estimation

3.2.1 Traditional COCOMO

COCOMO was first published in 1981 as a model for estimating effort, cost, and schedule for software projects (Boehm, 1981). References to this model typically call it COCOMO 81. In 1997 COCOMO II was developed and finally published in 2001 in the book Software Cost Estimation with COCOMO II. COCOMO II is the successor of COCOMO 81 and is better suited for estimating modern software development projects. It provides more support for modern software development processes and an updated project database. The need for the new model came as software development technology moved from mainframe and overnight batch processing to desktop development, code reusability and the use of off-the-shelf software components.

COCOMO consists of a hierarchy of three increasingly detailed and accurate forms. The first level, Basic COCOMO is good for quick, early, rough order of magnitude estimates of software costs, but its accuracy is limited due to its lack of factors to account for difference in project attributes (Cost Drivers). Intermediate COCOMO takes these Cost Drivers into account and Detailed COCOMO additionally accounts for the influence of individual project phases.
Basic COCOMO is a static, single-valued model that computes software development effort (and cost) as a function of program size expressed in estimated lines of code. COCOMO applies to three classes of software projects (Boehm et.al, 2000; Sunita Chulani, 2000):

- Organic projects - are relatively small, simple software projects in which small teams with good application experience work to a set of less than rigid requirements.
- Semi-detached projects - are intermediate (in size and complexity) software projects in which teams with mixed experience levels must meet a mix of rigid and less than rigid requirements.
- Embedded projects - are software projects that must be developed within a set of tight hardware, software, and operational constraints.

The basic COCOMO equations take the form

$$E=a_b(KLOC)^{b_b}$$
$$D=c_b(E)^{d_b}$$
$$P=E/D$$

Where E is the effort applied in person-months, D is the development time in chronological months, KLOC is the estimated number of delivered lines of code for the project (expressed in thousands), and P is the number of people required. The coefficients a_b, b_b, c_b and d_b are given in the following table.

Software project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Basic COCOMO is good for quick, early, rough order of magnitude estimates of software costs, but it does not account for differences in hardware constraints, personnel quality and experience, use of modern tools and techniques, and other project attributes known to have a significant influence on software costs, which limits its accuracy (Boehm et al.2000; Sunita Chulani, 2000).

3.2.2 Intermediate COCOMO

Intermediate COCOMO computes software development effort as function of program size and a set of "cost drivers" that include subjective assessment of product, hardware, personnel and project attributes. This extension considers a set of four "cost drivers", each with a number of subsidiary attributes:

- Product attributes
    - Required software reliability
    - Size of application database
    - Complexity of the product
- Hardware attributes
    - Run-time performance constraints
    - Memory constraints
    - Volatility of the virtual machine environment
    - Required turnabout time
- Personnel attributes
    - Analyst capability
    - Software engineering capability
    - Applications experience
    - Virtual machine experience
    - Programming language experience
- Project attributes
    - Use of software tools
    - Application of software engineering methods
    - Required development schedule

Each of the 15 attributes receives a rating on a six-point scale that ranges from "very low" to "extra high" (in importance or value). An effort multiplier from the table below applies to the rating. The product of all effort multipliers results in an *effort adjustment factor (EAF)*. Typical values for EAF range from 0.9 to 1.4.

	Ratings					
Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
Required software reliability	0.75	0.88	1.00	1.15	1.40	
Size of application database		0.94	1.00	1.08	1.16	
Complexity of the product	0.70	0.85	1.00	1.15	1.30	1.65
Hardware attributes						
Run-time performance constraints			1.00	1.11	1.30	1.66
Memory constraints			1.00	1.06	1.21	1.56
Volatility of the virtual machine environment		0.87	1.00	1.15	1.30	
Required turnabout time		0.87	1.00	1.07	1.15	
Personnel attributes						
Analyst capability	1.46	1.19	1.00	0.86	0.71	
Applications experience	1.29	1.13	1.00	0.91	0.82	
Software engineer capability	1.42	1.17	1.00	0.86	0.70	
Virtual machine experience	1.21	1.10	1.00	0.90		
Programming language experience	1.14	1.07	1.00	0.95		
Project attributes						
Use of software tools	1.24	1.10	1.00	0.91	0.82	
Application of software engineering methods	1.24	1.10	1.00	0.91	0.83	
Required development schedule	1.23	1.08	1.00	1.04	1.10	

The Intermediate COCOMO formula now takes the form:

$$\mathbf{E}=a_i(\mathbf{KLOC})^{(b_i)}.\mathbf{EAF}$$

where E is the effort applied in person-months, **KLOC** is the estimated number of thousands of delivered lines of code for the project, and **EAF** is the factor calculated above. The coefficient $\mathbf{a_i}$ and the exponent $\mathbf{b_i}$ are given in the next table.

Software project	$\mathbf{a_i}$	$\mathbf{b_i}$
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

The Development time **D** calculation uses **E** in the same way as in the Basic COCOMO

3.2.3 Detailed COCOMO

Detailed COCOMO - incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process

3.2.4 COnstructive COst MOdel version II (COCOMO II)

COCOMO II can be used for the following major decision situations (Boehm et al. 2000)

- Making investment or other financial decisions involving a software development effort
- Setting project budgets and schedules as a basis for planning and control
- Deciding on or negotiating tradeoffs among software cost, schedule, functionality, performance or quality factors
- Making software cost and schedule risk management decisions
- Deciding which parts of a software system to develop, reuse, lease, or purchase
- Making legacy software inventory decisions: what parts to modify, phase out, outsource, etc
- Setting mixed investment strategies to improve organization's software capability, via reuse, tools, process maturity, outsourcing, etc
- Deciding how to implement a process improvement strategy, such as that provided in the SEI CMM

The full COCOMO II model includes three stages.
Stage 1 supports estimation of prototyping or applications composition efforts.
Stage 2 supports estimation in the Early Design stage of a project, when less is known about the project's cost drivers. Stage 3 supports estimation in the Post-Architecture stage of a project.
This version of USC COCOMO II implements stage 3 formulas to estimate the effort, schedule, and cost required to develop a software product. It also provides the breakdown of effort and schedule into software life-cycle phases and activities from both the Waterfall model and the Mbase Model. The Mbase model is fully described in Software Cost Estimation with COCOMO II.

3.2.5 COCOTS

COCOTS is the acronym for the COnstructive COTS integration cost model, where COTS in turn is short for commercial-off-the-shelf, and refers to those pre-built, commercially available software components that are becoming ever more important in the creation of new software systems.

The rationale for building COTS-containing systems is that they will involve less development time by taking advantage of existing, market proven, vendor supported products, thereby reducing overall system development costs (Abs et al., 2000). But there are

two defining characteristics of COTS software, and they drive the whole COTS usage process:
1) The COTS product source code is not available to the application developer, and
2) The future evolution of the COTS product is not under the control of the application developer.
Because of these characteristics, there is a trade-off in using the COTS approach in that new software development time can indeed be reduced, but generally at the cost of an increase in software component integration work. The long term cost implications of adopting the COTS approach are even more profound, because considering COTS components for a new system means adopting a new way of doing business till the retirement of that system. This is because COTS software is not static; it continually evolves in response to the market, and the system developer must adopt methodologies that cost-effectively manage the use of those evolving components.

3.2.5.1 Relation to COCOMO II

COCOMO II creates effort and schedule estimates for software systems built using a variety of techniques or approaches. The first and primary approach modeled by COCOMO is the use of system components that are built from scratch, that is, new code. But COCOMO II also allows to model the case in which system components are built out of pre-existing source code that is modified or adapted to current purpose, i.e., reuse code.
What COCOMO II currently does not model is that case in which there is no access to a pre-existing component's source code. We have to take the component as is, working only with its executable file, and at most are able to build a software shell around the component to adapt its functionality to our needs.
This is where COCOTS comes in. COCOTS are being designed specifically to model the unique conditions and practices highlighted in the preceding section that obtain when we have to incorporate COTS components into the design of our larger system.

3.2.5.2 COCOTS Model Overview

COCOTS at the moment are composed of four related sub models, each addressing individually what we have identified as the four primary sources of COTS software integration costs (Grady, 1997).
Initial integration costs are due to the effort needed to perform (1) candidate COTS component assessment, (2) COTS component tailoring, (3) the development and testing of any integration or "glue" code (sometimes called “glue ware” or “binding” code) needed to plug a COTS component into a larger system, and (4) increased system level programming and testing due to volatility in incorporated COTS components.
The following figure illustrates how the modeling of these effort sources in COCOTS is related to effort modeled by COCOMO II.

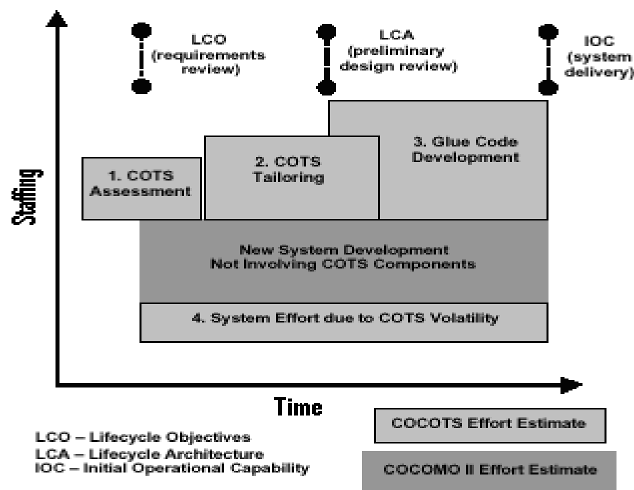

Fig. 2. Sources of Effort (Abts.et.al, 2000b)

The figure represents the total effort to build a software system out of a mix of new code and COTS components as estimated by a combination of COCOMO II and COCOTS. The central block in the diagram indicates the COCOMO II estimate, that is, the effort associated with any newly developed software in the system. The smaller, exterior blocks indicate COCOTS estimates, that effort associated with the COTS components in the system. The relative sizes of the various blocks in this figure is a function of the number of COTS components relative to the amount of new code in the system, and of the nature of the COTS component integration efforts themselves. The more complex the tailoring and/or glue code-writing efforts, the larger these blocks will be relative to the assessment block.

4. Software Metrics

Software development is a complex undertaking. Successful management requires good management skills and good management information. Software metrics are an integral part of the state-of the-practice in software engineering. A sound software metrics program can contribute significantly to providing great management information. Successful metrics programs must provide sound management information to better understand, track, control and predict software projects, processes and products while ensuring low- cost, simplicity, accuracy, and appropriateness.

According to Goodman, software metrics as, "The continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products" (Goodman,1993).

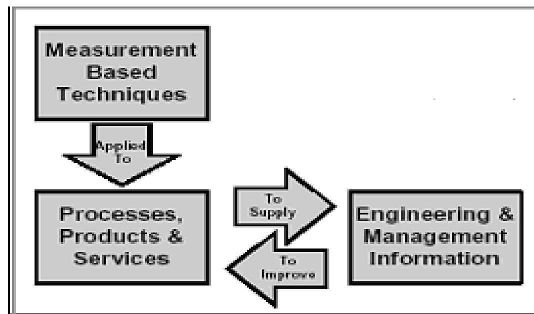

Fig. 3. What are Software Metrics?

Fig. 3 illustrates that metrics can provide the information needed by engineers for technical decisions as well as information required by management.
A software metrics program has many purposes such as cost and schedule estimation, identifying and controlling of risks, evaluation of bids, resource allocation, requirements management, predicting schedules, reducing defects, assessing progress, and improving processes.

4.1 The Keys to Setting Up a Successful Software Metrics Program

The keys to setting up a successful software metrics program are (Chidamber &Kemerer, 1994):

1. Keep the metrics simple keeping in mind that the goal of a metrics program is management not measurement.
2. Understand the different types of metrics indicators.

   E.g: a. Metrics designed to provide an accurate assessment of complicated development path.

   b. Metrics designed to provide an early identification of processes that have broken down.

   c. Metrics designed to provide an advanced warning of trouble ahead. All types are important, but they require different management attention.

   d. Metrics designed to provide an accurate picture of where the project is with regard to cost and schedule.
3. Keep the metrics measurement cycle short.
4. Use a balanced set of metrics. Unbalanced metrics programs can fail because they drive an organization into undesirable behavior.
5. Keep the metrics collection cost down. Normally, software metrics collection program costs should not exceed 5 percent of the development cost. Automated tools can be implemented to collect many metrics.
6. Focus on a small set of important metrics. Some metrics programs suffer from a tendency to collect more and more information over time. Additional metrics do not always provide sufficient additional information to justify their expense.

Keep in mind that metrics will not replace management, and are most effectively used to provide data about potential problem areas to focus management attention (Poulin & Caruso, 1993).

4.2 Deciding and Managing the Metrics

There are literally thousands of possible software metrics to collect and possible things to measure about software development. There are many books and training programs available about software metrics. Later in this document, I am trying to provide a "minimum set" of top-level metrics suitable for reuse program,
For each metric, one must consider *(*Henderson-Sellers, 1996*)*:

1. What are you trying to manage with this metric? Each metric must relate to a specific management area of interest in a direct way
2. What does this metric measure? Exactly what does this metric count?
3. If your organization optimized this metric alone, what other important aspects of your software development phases would be affected?
4. How hard/expensive is it to collect this information? This is where you actually get to identify whether collection of this metric is worth the effort.
5. Does the collection of this metric interact with (or interfere with) other business processes?
6. How accurate will the information be after you collect it?
7. Can this management interest area be measured by other metrics? What alternatives to this metric exist?

Metrics are not useful if they can not be easily reviewed, analyzed for trends, compared to each other, and displayed in a variety of manners. Periodic review of existing metrics against the points mentioned above is recommended.
Be sure to take advantage of free metrics tools available where they are appropriate (example sources of tools include: Software Technology Support Center (STSC), the Software Engineering Institute (SEI), the Software Productivity Consortium (SPC), and the Software Program Managers Network (SPMN)).With this guidance in mind, let's turn to selecting the proper metrics to measure reuse cost and effort.

4.3 Measuring Software Reuse Cost

Economic considerations are at the center of any discussion of software reuse. Economic models and software metrics are needed that quantify the costs and benefits of reuse Models for software reuse economics try to help us answer the question, “when is it worthwhile to incorporate reusable components into a development and when is custom development without reuse preferable?” *(*Heinemann &Councill, 2001*)*.Further, different technical approaches to reuse have different investment and return on investment profiles (Poulin & Caruso, 1993; Frakes & Terry, 1994).Generally metrics can be categorized into two namely product metrics, which determine the characteristics of components and process metrics, which measure time, cost etc.Only recently the researchers started to tackle this problem (Frakes & Terry, 1996; Barnes & Bollinger, 1991; Mili et.al, 1995). But even such studies couldn’t help to convince the management to understand the advantage of reuse. Most

existing software engineering economic models need to be customized to each specific reuse business. Several authors have modified the cost models that are today used to estimate time and effort and for the development both of components and of applications using components (Malan & Wentzel, 1993; Poulin, 1994; Boehm & Papaccio, 1988).Because high levels of reuse can reduce the overall cost and time to deliver applications, the extra funding and time can be directed to several alternative projects.
In the following section, the implications of various approaches for software reuse in the organizations are discussed and proposed few economic models for cost analysis.

4.4 Software Reuse Cost Estimation Models

We can categorize the type of reuse in the context of cost estimation as follows (Barnes & Bollinger,1991):

i) Component Reuse without Modification
ii) Component Reuse with modification

In the case of component reuse without modification, the average cost of developing using reusable components can be formulated as follows:

$$\text{Cost}_{\text{search}} + (1-p)*\text{Development}_{\text{no-reuse}} \tag{1}$$

where $\text{Cost}_{\text{search}}$ is the cost of performing a search operation, $\text{Development}_{\text{no-reuse}}$ is the cost of developing without reuse (i.e., the cost of developing the component from scratch) and p is the probability that the component is found in the component library. It is observed that the reuse option would be preferable only if:

$$\text{Cost}_{\text{search}} + (1-p)*\text{Development}_{\text{no-reuse}} < \text{Development}_{\text{no-reuse}} \tag{2}$$

In the case of component reuse with modification, the average cost of developing using reusable components can be formulated as follows:

$$\text{Cost}_{\text{search}} + \text{Cost}_{\text{adapt}} + (1-p)*\text{Development}_{\text{no-reuse}} \tag{3}$$

where $\text{Cost}_{\text{search}}$ is the cost of performing a search operation (the cost depends on the whether the search is a manual search or search using a search tool), $\text{Cost}_{\text{adapt}}$ is the cost required to adapt the component, $\text{Development}_{\text{no-reuse}}$ and p means the same as in the case of equation (1). It is observed that the reuse option would be preferable only if:

$$\text{Cost}_{\text{search}} + \text{Cost}_{\text{adapt}} + (1-p)*\text{Development}_{\text{no-reuse}} < \text{Development}_{\text{no-reuse}} \tag{4}$$

In both the cases, the cost saving due to reuse can be formulated using a simple equation:

$$\text{Cost}_{\text{saved}} = \text{Cost}_{\text{no-reuse}} - \text{cost}_{\text{reuse}} \tag{5}$$

In addition to the above costs, we should also consider some overhead costs associated with reuse include (Jasmine & Vasantha, 2008a):

- Domain analysis (Balda & Gustafson,1990;Pressman,2001)
- Increased documentation to facilitate reuse

- Maintenance and enhancement of reuse artifacts (documents and components)
- Royalties and licenses for externally acquired components
- Creation (or acquisition) and operation of a reuse repository (If the decision is to build a reusable component, then the cost of initial development and also the expected usage frequency of the component also should be considered (Mili.et.al,1995)
- Training of personnel in design for reuses and designs with reuse.

To maximize reuse profits, by analyzing process, organizational and technical aspects with reduced asset development cost and management cost, we have to consider the following points

i) When and Where Capital investment is to be made

Two approaches were observed namely, proactive and reactive. 80% of identified population supported proactive approach and 20% supported reactive approach. If the domain is stable, where the product features can be predicted, organizations can go for upfront investment to develop reusable assets (proactive approach). If the domain is unstable, reusable assets can be developed as when required (reactive approach). This approach may result in reengineering and retrofitting existing products with reusable assets, if there is no common architectural basis.

ii) Whether to go for a dedicated team for development/distribute/maintain assets or not and associated costs involved.

Again the Cost no-reuse and cost reuse depends on the % size of parts (components) reused and % of parts (components) not reused.

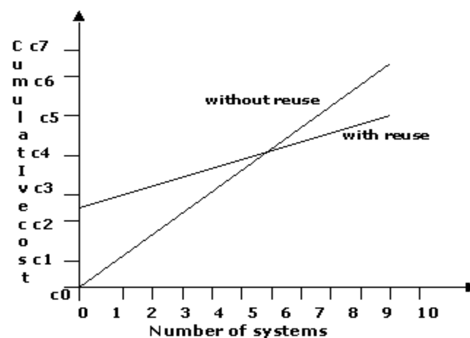

Fig. 4. Cumulative costs of software systems without reuse vs. with reuse

Fig 4 illustrates that for a reuse oriented software development; there will be an initial cost increase. Then gradually cost will decrease due to reusing the same component again and again across similar products.

Thank You for previewing this eBook

You can read the full version of this eBook in different formats:

- HTML (Free /Available to everyone)
- PDF / TXT (Available to V.I.P. members. Free Standard members can access up to 5 PDF/TXT eBooks per month each month)
- Epub & Mobipocket (Exclusive to V.I.P. members)

To download this full book, simply select the format you desire below