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 'darwinistic' 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.

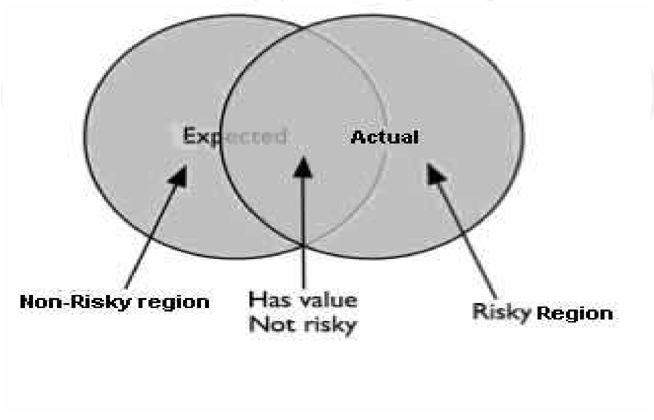

Fig. 1. Prediction region for acceptable values

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.

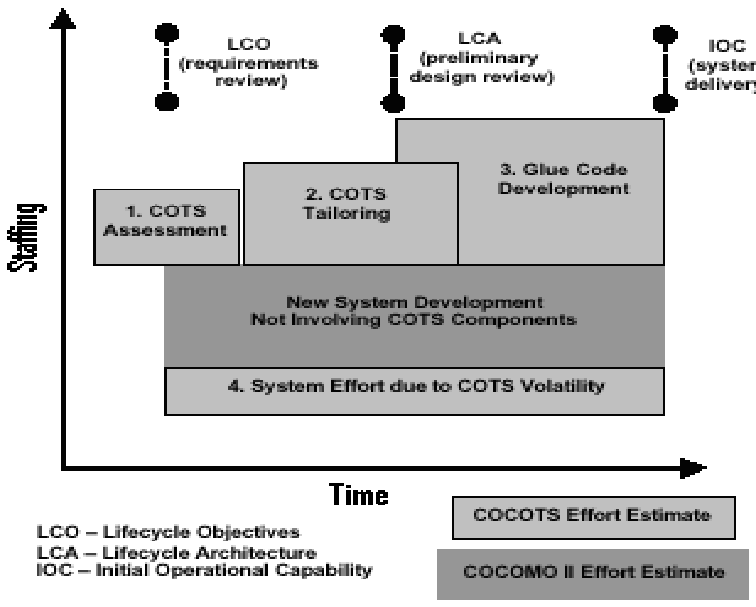
LCO (requirements review)
LCA (preliminary design review)
IOC (system delivery)
Staffing
1. COTS Assessment
2. COTS Tailoring
3. Glue Code Development
New System Development Not Involving COTS Components
4. System Effort due to COTS Volatility
Time
LCO – Lifecycle Objectives
LCA – Lifecycle Architecture
IOC – Initial Operational Capability
COCOTS Effort Estimate
COCOMO II Effort Estimate

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).

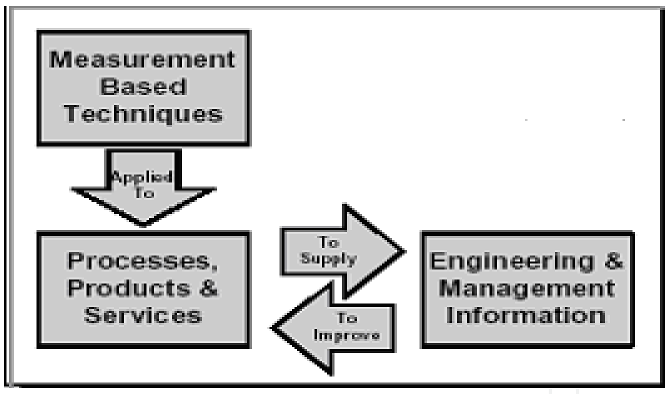
Measurement Based Techniques
Applied To
Processes, Products & Services
To Supply
To Improve
Engineering & Management Information
Fig. 3. What are Software Metrics?

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:

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

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:

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

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

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

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:

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

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

$$\text{Cost}_{\text{saved}} = \text{Cost}_{\text{no-reuse}} - \text{cost}_{\text{reuse}} \qquad (5)$$

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.

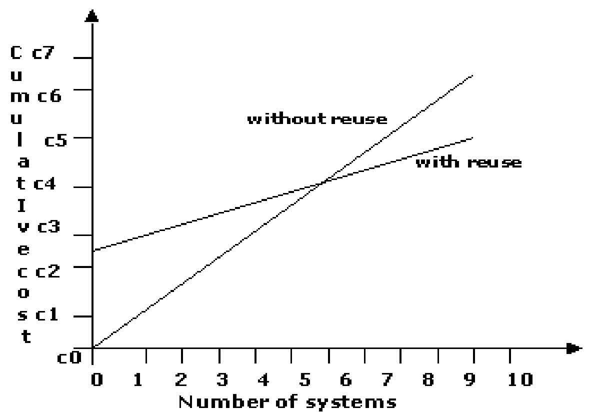

Fig. 4. Cumulative costs of software systems without reuse vs. with reuse

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.

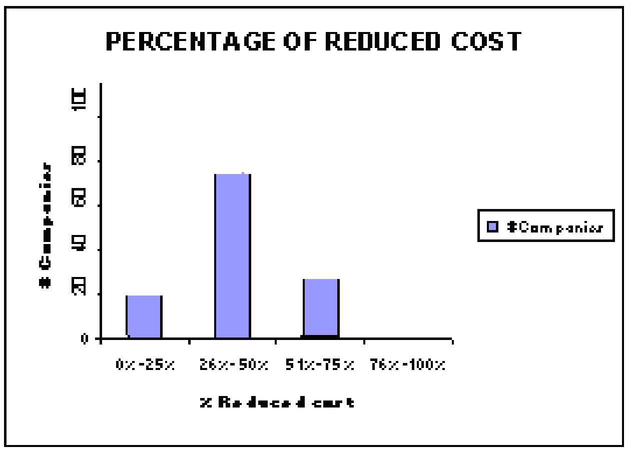

Fig. 5. Percentage of reduced cost of software products due to reuse

Fig 5 illustrates the percentage of reduced cost in the surveyed software companies. The % of cost reduction due to reuse is observed as three ranges such as between 0% and 25%, 26% and 50% & 51% and 75%. Also none of the companies are completely saved their development cost due to reuse.

4.5 Measuring software reuse effort

4.5.1 Suggested estimation Model

A common approach to estimate effort is to make it a function of project size and equation of effort is considered as follows:

$$\text{Effort}=a*\text{size}^{\,b} \tag{6}$$

where a and b are constants (Pressman, 2001).A similar study on smaller projects showed that the data fits a straight line quite well, and the equation is of the form:

$$\text{Effort}=a* \text{size} + b \tag{7}$$

where a and b are constants that are obtained by analysis of data of past projects. Where a is termed the effort adjustment factor and b the scale factor.
Taking into consideration of reusable components these two equations can be modified as follows (Jasmine & Vasantha, 2008b):

$$\text{Effort} = a1*(\text{size of Part}_{\text{reuse}})^{b1} + a2*(\text{size of Part}_{\text{no-reuse}})^{b2} \tag{8}$$

$$\text{Effort} = (a1*(\text{size of Part}_{\text{reuse}}) +b1) + (a2*(\text{size of Part}_{\text{no- reuse}}) +b2) \tag{9}$$

where a1, b1 a2, b2 are constants, part $_{\text{reuse}}$ is part developed from reusable components and part$_{\text{no-reuse}}$ is part developed without reusable components.

4.5.2 Incremental development and effort estimation

The effort estimation model used here relates the development effort (in project per-son-months, PM) to software size (in thousands of delivered source lines of code or in number of function points) as by replacing effort by y and size by x the equation (5) becomes

$$y = ax^b \tag{10}$$

In incremental development the target system S is delivered in n stages (Mili.et.al, 1995). The development is seen as a sequence of component integration C1, C2… Cn where each component is a functionally independent system delivered to a customer and the functionality increases with each delivery with one or more component integrations. C1 is considered to be the start of a new system .At the end of the development Cn is equivalent to the target system, S.
The size of the target system S is represented by x. In a one-off (non-incremental) development this effort is estimated with equation (10).
A nominal size of increment i (i.e. the additional functionality, or code, needed to move from Ci-1 to Ci) is represented by n_i and the totality of the increments make up the target system so that

$$x = \sum_i \mathsf{n}_i \tag{11}$$

The effective size, x_i, of each increment for effort estimation purposes is expressed as

$$x_i = \mathsf{n}_i + \mathsf{o}_i \mathsf{n}_{i-1} \tag{12}$$

where the parameter O reflects the overhead associated with the previous increment. For example if O has a value of 0.25, it corresponds to 25% overhead.
The development effort in enhancing system from stage Ci-1 to Ci involves developing an increment of effective size x_i and the effort is thus estimated by using expression (10):

$$y_i = a_i x_i^{b_i} \tag{13}$$

giving

$$\sum_i a_i x_i^{b_i} \tag{14}$$

Replacing x_i by $n_i+o_in_{i-1}$,
The final equation for y_i is as follows

$$\sum_i a_i(n_i + o_i n_{i-1})^{b_i}$$ (Jasmine & Vasantha, 2008b) (15)

In a similar manner equation (6) also can be modified.
In incremental development (except for agile development) significant one-time tasks need to be undertaken up front: product design for incremental systems must be well thought out and open-ended so that each additional increment can be incorporated into the architecture defined initially for the whole product. The architecture of the solution needs to allow for breakdown into increments thereby enabling staged development. This requires an upfront effort (encompassing core capability and overall architecture included in the first release), which is over and above the effort needed to develop each increment (Laraman&Basili, 2003).The initial effort can be of considerable magnitude, especially if domain architecting and reuse considerations are taken into account.

Models for software reuse economics helps the organizations to decide When is it worthwhile to incorporate reusable components into a development and when is to perform custom development without reuse. It is observed from the study that existing software engineering economic models need to be adjusted to include reuse and customized to each specific reuse business. An extensive work relating more elaborate software structure reuse metrics and economical factors, with a wide variety of representative set of industrial organizations is still necessary to standardize the reuse cost metrics field. Future research can be performed in these directions.

5. Systematic reuse and economic feasibility

Generally, the knowledge about the problem domain is realized through a set of components that are reusable within a formerly defined framework. A practical problem in defining the problem domain is that the domain knowledge usually isn't immediately available. An essential difficulty herewith is the impossibility to predict all possible variations and evolutions of the software (and the problem domain) beforehand.

Systematic reuse should not only recognize the need for a reusable asset to evolve both during its initial design and when it is being reused, it should actually advocate the development of a methodology for managing change in the process of engineering reusable software. The development of reusable assets is inherently an evolutionary process. A reuser can only gain insights in the qualities of reusable assets by actually reusing them. A provider can only improve the qualities of assets if the experience of reuse is fed back to him. Successful assets can have a long life span and thus need to evolve and adapt to new reusers and their requirements. The inability to do so turns a reusable asset into legacy. Such iterative development is also important because it allows the construction of reusable assets in a bottom-up fashion. This is crucial for reuse to become economically feasible: it allows finding a delicate balance between the longer term investments needed for constructing reusable assets and the need to meet shorter term (customer) deadlines. To be able to leverage on the investment made in building an asset, reusers must be able to benefit from future improvements of the assets they reuse: proper evolution of reused assets should not invalidate previous reuse. In a similar vein, reuse should go beyond the act of copying out code fragments and adapting them to current requirements without regard for the evolution of the reused fragments. This implies the management of some kind of consistency in the evolution of reusable software, to prohibit different versions of a reusable asset from propagating through different applications. While systematic reuse should present an opportunity to reduce maintenance effort, a proliferation of versions actually increases it, as older versions of an asset behave differently than newer versions. The absence of change management mechanisms is recognized as an important inhibitor to successful reuse.

To reconcile the bottom-up approach of iterative development with reuse with the top-down approach of systematic reuse (development for reuse), a methodology that combines the best of both worlds by introducing systematic reuse in the object-oriented software engineering process can be adopted. My conjecture is that incrementally building reusable assets requires a strong co-operation between providers of reusable assets and asset reusers.

5.1 Product-Line Architectures and Reusable Assets

Product family/line engineering is all about reusing components and structures as much as possible. This method creates an underlying architecture of an organizations product platform, one that is based on commonality and similarity. The various product variants can be derived from the basic product family, which creates the opportunity to reuse and differentiate on products in the family. It focuses on the process of engineering new products in such a way that it is possible to reuse product components and apply variability with decreased costs and time.
A product line involves core asset development and product development using the core assets, both under the aegis of technical and organizational management. Core asset development and product development from the core assets can occur in either order: new products are built from core assets, or core assets are extracted from existing products. Often, products and core assets are built in concert with each other. The following figure illustrates these activities.

The Three Essential Activities for Software Product Lines are (Bosch, 1998b)
1. Core Asset Development
2. Product development
3. Management

All three are linked together and they are all essential, inextricably linked, and highly iterative, and can occur in any order .Core assets are used to develop products, but that revisions of existing core assets or even new core assets might, and most often do, evolve out of product development.

5.1.2 What are core assets?

In some contexts, already existing products are mined for generic assets–perhaps a requirements specification, an architecture, or software components–which are then

migrated into the product line's core asset base. In other cases, the core assets may be developed or procured for later use in the production of products.
There is a strong feedback loop between the core assets and the products. Core assets are refreshed as new products are developed. The use of core assets is tracked, and the results are fed back to the core asset development activity. In addition, the value of the core assets is realized through the products that are developed from them. As a result, the core assets are made more generic by considering potential new products on the horizon. There is a constant need for strong, visionary management to invest resources in the development and sustainment of the core assets. Management must also precipitate the cultural change to view new products in the context of the available core assets. Either new products must align with the existing core assets, or the core assets must be updated to reflect the new products that are being marketed. Iteration is inherent in product line activities–that is, in turning out core assets and products and in coordinating the two.

5.1.3 Problems related to Reusable assets

Based on the interviews and other documentation collected from the various organizations as part of the study, I have identified a number of problems related to reusable assets that I believe to have relevance in a wider context than just these organizations. The problems are categorized into three categories, related to multiple versions of assets, dependencies between assets and the use of assets in new contexts.

5.1.3.1 Multiple versions of assets

Product-line architectures have associated reusable assets that implement the functionality of architectural component. These assets can be very large and contain up to a hundred KLOC or more. Consequently, these assets represent considerable investments, multiple man-years in certain cases. Therefore, it was surprising to identify that in some cases, the interviewed companies maintained multiple versions (implementations) of assets in parallel. One can identify at least four situations where multiple versions are introduced (Dikel et.al, 1997; Bass et.al, 1997) which are mentioned below.

- **Conflicting quality requirements**: The reusable assets that are part of the product line are generally optimized for particular quality attributes, e.g., performance or code size. Different products in the product-line, even though they require the same functionality, may have conflicting quality requirements. These requirements may have so high priority that no single component can fulfill both. The reusability of the affected asset is then restricted to only one or a few of the products while other products require another implementation of the same functionality.
- **Variability implemented through versions**: Certain types of variability are difficult to implement through configuration since the effect of a variation spreads out throughout the reusable asset e.g., operating system, for an asset. Although it might be possible to implement all variability through, often it is decided to maintain two different versions.
- **High-end versus low-end products**: The reusable asset should contain all functionality required by the products in the product-line, including the high-end products. The problem is that low-end products, generally requiring a restricted subset of the functionality, pay for the unused functionality in terms of code size

and complex interfaces. Whenever the hardware cost play an important role in the product price, the software engineers may be forced to create a low-end, scaled-down version of the asset to minimize the overhead for low-end Products.

- **Asset evolution**: When the business units are responsible for asset evolution, assets are sometimes extended with very product-specific code or code only tested for one of the products in the product-line (Jacobson et.al,2000). The problems caused by this create a tendency within the affected business units to create their own copy of the asset and maintain it for their own product only. This minimizes the dependency on the shared product-line architecture and solves the problems in the short term, but in the long term it generally does not pay off. There will be several instances of cases where business units had to rework considerable parts of their code to incorporate a new version of the evolved shared asset that contained functionality that needed to be incorporated in their product also.

5.1.3.2 Dependencies between assets

Since the reusable assets are all part of a product-line architecture, they tend to have dependencies between them. Although dependencies between assets are necessary, assets often have dependencies that could have been avoided by another modularization of the system or a more careful asset design. From the examples from the surveyed companies, I have learned that the initial design of assets generally defines a small set of required and explicitly defined dependencies. It is often during evolution of assets that unwanted dependencies are created. Addition of new functionality may require extension of more than one asset and in the process often dependencies are created between the involved assets to implement the functionality. These new dependencies could often have been avoided by another decomposition of the architecture and have a tendency to be implicit, in that their documentation is often minimal and the software engineer encounters the dependency late in the development process. Dependencies in general, but especially implicit dependencies, reduce the reusability of assets in different contexts, but also complicate the evolution of assets. Based on my research at various software companies, I have identified the following situations where new, often implicit, dependencies are introduced:

- **Component decomposition** With the development of the product-line architecture generally also the size of the reusable assets increases. With the increasing size of asset components, there is a point where a component needs to be split into two components. These two components, initially, have numerous relations to each other, but even after some redesign often several dependencies remain because the initial design did not modularize the behavior captured by the two components.
- **Extensions cover multiple assets**: During implementation of the extension, it is very natural to add dependencies between the affected assets since one is working on functionality that is perceived as one piece, even though it is divided over multiple assets. By separating the authorization access from the two assets and representing as a single asset, we can decrease the dependencies.

5.1.3.3 Assets in new contexts

Since assets represent considerable investments, it is mandatory to use assets in as many products and domains as possible to save money. However, the new context differs in one or more aspects from the old context, causing a need for the asset to be changed in order to

fit. Two main issues in the use of assets in new context can be identified (Bass et.al, 1997; Bosch, 1998b):
• **Mixed behavior**: An asset is developed for a particular domain, product category, operating context and set of driving quality requirements. Consequently, it often proves to be hard to apply the asset in different domains, products or operating contexts. The design of assets often hardwires design decisions concerning these aspects unless the type of variability is known and required at design time.
E.g., Assets used for fire-alarm system and intruder-alarm systems due to the similarities in the domains
• **Design for required variability**: A reusable asset often requires new variability dimensions. So assets should be designed so that the introduction of new variability requires minimal effort.
E.g, fire-alarm and intruder alarm system share, to a large extent, the same operating context and quality requirements. Since the fire-alarm framework is designed for its domain and the intruder alarm domain has different requirements and concepts, one is forced to introduce variability for application domain functionality.

5.2. Cause Analysis for the Identified problems

The problems discussed in the previous section present an overview over the issues surrounding the use of reusable assets in product-line architecture.

The following are the identified primary underlying causes for these problems. These causes are discussed in the following section (Bosch, 1998b).

5.2.1 Early intertwining of functionality

The early intertwining of functionality is a primary cause to several of the problems discussed in the previous section. Multiple versions of assets are required because the different categories of functionality cannot be separated in the implementation and implemented through variability. The use of layers in asset design separating operating context dependent from context independent functionality avoids the mixing. Also, some design patterns (Gamma et al., 1994; Buschmann et.al, 1996) support this.

5.2.2 Organization

Having explicit groups for domain and application engineering requires a relatively large software development department consisting of at least fifty to a hundred engineers. Several of the problems discussed earlier can be related to the lack of independent domain engineering. Business units focus on their own quality attributes and design for achieving those during asset extension. Because of that, multiple versions of assets may be created where a domain engineering unit may have found solutions allowing for a single version. In addition, asset extension without sufficient focus on the product-line as a whole may introduce more dependencies than strictly necessary, complicating the use of assets as well as the reuse of assets in new contexts.
Solutions exist to minimize the negative effects of organizational structures. Asset redesigns are performed when a consensus is present that an asset needs to be reorganized. During an

asset redesign, the software architects from the business units using the asset gather to redesign the asset in order to improve its structure.

5.2.3 Time To Markert

A third important cause for the problems related to reusable assets at the interviewed companies is the time-to-market (TTM) pressure. The problem most companies are dealing with is that products appearing late on the market will lead to diminished market share or, in the worst case, to no market penetration at all. Sacrificing some time-to-market for one product may lead to considerable improvements for subsequent products, but this is generally not appreciated. To address the problems resulting from TTM pressure, it is important for software development organizations to identify the development of product-line architecture with associated assets is a strategic issue and decisions should be taken at the appropriate level. The consequences for the time-to-market of products under development should be balanced against the future returns.

5.2.4 Economic models

Reusable assets may represent investments up to several man years of implementation effort. For most companies, such an asset represents a considerable amount of capital, but both engineers and management are not always aware of that. For instance, an increasing number of dependencies between assets is a sign of accelerated aging of software and, in effect, decreases the value of the assets. Economic models are necessary to visualize the effects of quick fixes causing increased dependencies, in order to establish the economic losses of these dependencies versus the time-to-market requirements. The lack of suitable economic models for these influences several of the identified problems.

There is a need for economic models in two situations. First, models for calculating the economic value of asset, based on the investment (man hours) but also on the value of the asset for future product development and/or for an external market. In addition, models visualizing the effects of various types of changes and extensions on the asset values are also required.

5.3 Some Issues and considerations for product-line approach

In the context of product line approach, the following issues to be considered (Griss et. al, 1993; Basili et.al, 1994).

- Business
    - Number of new starts subscribing to a software product line may be insufficient to offset the costs of building/maintaining product line
    - Incentive must be provided to the contractor community to engage in product line asset development
    - Contractor liability must be addressed and resolved
- Technical
    - Need for software engineering practices and processes that apply technology to create and evolve product line assets and products

  - Managing the evolution of product line architectures
- Organizational
  - Importance of infrastructure and cross organizational cooperation
  - Need for strong visionary management to invest resources into the development/containment of product line architectures

6. Reuse Capability maturity Model (RCMM)

The RCMM was developed in the early 1990's at the Software Productivity Consortium (SPC) as part of a systematic approach to reuse adoption. It is an effective guide primarily to help organizations identify opportunities for improving a reuse-based process. It also provides criteria by which an organization can target a level of reuse capability that matches its particular needs and capabilities (SPC-CMC, 1993).However, the interaction between these two purposes and with the CMM has not been fully understood. Now, our study with various software organizations established that the critical success factors defined by the RCMM can be entirely partitioned according to these two purposes.

Capability Maturity Model is a reference model of mature practices in a specified discipline, used to assess a group's capability to perform that discipline (Craig& Allgood, 2001)
• CMMs differ by
–Discipline (software, systems, acquisition, etc.)
–Structure (staged versus continuous)
–How Maturity is Defined (process improvement path)
–How Capability is Defined (institutionalization)
As per Carnegle Mellon Software Engineering institute's study, CMMI transition from traditional approach to reuse oriented CMMI is a technology transition. Because CMMI is a process technology (Garcia, 2003).

CMMI will require
- Development of new routines(procedures)
- Modification in the norms and beliefs of the organization members

6.1 Need of Technology Transition Practices (TTP) with CMMI transition

TTP is business unit within the Technology Transition Services Directorate focused on enabling organizations to build their own capability for managed, accelerated and affordable software technology transition.TTP works with all SEI software technologies.TTP's current focus is on working primarily with technology developers and deployers. The role of TTP is to work with CMMI team to monitor /refine transition strategy (CMU/SEI, 1993; Garcia, 2003).

6.2 Key elements in successful technology transition

1. Understanding the goals of different roles involved in the transition process
2. Understanding the characteristics of the technology

3. Understanding what is needed to make the technology work.
4. Identifying and mitigating the identified risks

6.3 Different roles in Technology transition

1. Technology developers :Those who create new technologies

E.g. SEI initiatives, Commercial product motivation teams

2. Technology acquirers Those who determine which technologies will be used to support their own system development efforts

E.g. Individual acquisition program offices and corporate business units

3. Technology deployers Those who determine which technologies will be used to support their own system development efforts

E.g. SEI transition partners, military organizations like STSC

4. Technology adopters: The organization or group that will actually be using a new technology

E.g. War fighter units in military, organizations adopting a new maturity model

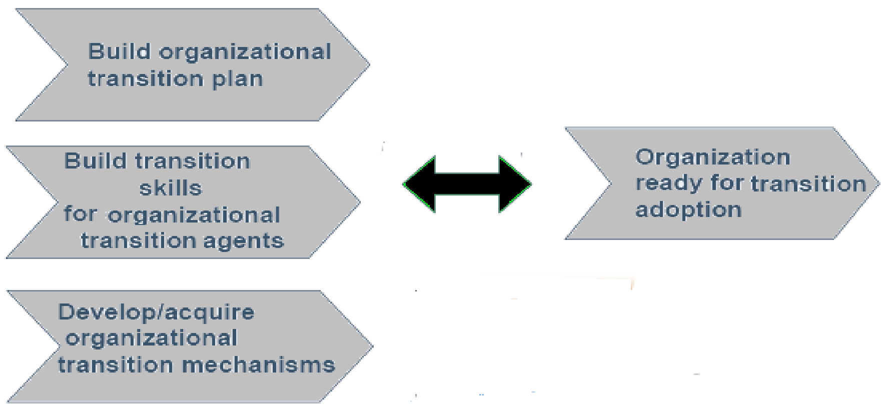
Build organizational transition plan
Build transition skills for organizational transition agents
Develop/acquire organizational transition mechanisms
Organization ready for transition adoption

Fig. 6. Adaption and Planning for Transition Adopting Organization

6.4 Motivation for Reuse CMM

Motivation for developing the Reuse Maturity Model comes from observing the enormous impact the Software Engineering Institute's software Capability Maturity Model has had. The SEI has focused attention on process issues in a most remarkable way by examining characteristic engineering practices and providing a means to classify an organization into one of five maturity levels (CMU/SEI, 1993; Garcia, 2003). Moreover, the SEI has established the principle in people's minds that the quality of a software product is dictated by the quality of the processes used to develop that product. Furthermore, the SEI has convinced executive managers that process improvement must come a step at a time, by laying a foundation for process improvement on which other improvement activities can build. Through my Reuse Maturity Model, I hope that achieving reuse requires a comprehensive approach.

From the survey of software companies, a numerous obstacles are identified that must be overcome in order to achieve high levels of reuse, are listed below:

Cultural: Incentives and management backing must be put in place and (``Not Invented Here'') syndrome must be eliminated.

Institutional: A corporate-wide forum is needed to identify product development cycle where reuse concerns can always be raised and resolved.

Financial: The costs and benefits must be understood for a product life cycle based on a "Design for Reuse" philosophy. Reusable work-products must be viewed as capital assets.

Technical: Proper mechanisms are needed to ensure that guidelines, techniques, and standards for making things reusable are developed and followed.

Legal: Negotiations must be undertaken to determine how to retain rights to components developed under customer contract and recover costs in a reuse context. Mechanisms will be needed for payment and collection of royalties for use and reuse in the commercial arena.

Out of the hundred industrial organizations considered, I could see many Level 1 Reuse organizations, only a handful of companies at the intermediate levels, and only hypothesize what a Level 5 Reuse company would look like, I expect some significant revisions are needed to the model. An additional step which augments the reuse maturity questionnaire and by organizing subsets of the questions to address each level of reuse maturity separately is required to objectively measure the Progress toward reuse process improvement.

6.5 How to achieve effective reuse

To provide organizations with detailed guidance on how to achieve effective reuse, the Reuse-driven Software Processes (RSP) methodology was also developed at SPC in the early 1990's. All RSP processes consist of two distinct lifecycle activities of domain engineering and application engineering. The conceptual basis of any RSP process is the formalization of commonalities and variabilities that characterize a set of similar products to represent a product family and an associated process for deriving instance products to meet diverse and changing customer needs.

Other authors have proposed different models to structure the breadth of reuse involvement provided by an organization. Among them one was proposed by (Koltun & Hudson, 1991), that proposed by the Software productivity Consortium and that used in the UE project REBOOT (Reuse Based on Object oriented Techniques)

The model proposed by Koltun and Hudson, five maturity levels are defined for reuse:

1- Initial Chaotic
2- Monitored
3- Co-coordinated
4- Planned
5- Ingrained

The criteria that permit the evaluation of the level of each organization in the model are: Motivation, Planning for reuse, Breadth of reuse involvement, Responsibility for making reuse happen, process by which reuse is leveraged, reuse inventory, classification activity, Technology support, metrics and legal considerations.
The model suggested by Llorens Morillo et. al, is based on the monitoring of three different factors

- Repository structure
- Software development architecture
- Administrative management

Each factor encompasses a certain part of the reuse environment, covering the following areas of control.

Repository structure deals with information representation of the available information in a manner to reuse in the future, classification of existing components and suitable recovery techniques to obtain wherever necessary, and management of authorizing, rejecting and modification of existing components and automated announcement of incorporation or modification of components (Topaloglu et. al, 1996)

Software Development architecture includes developing the architecture according to its orientation towards reuse, type of reuse systematically achieved by the organization and component testing.

Under administrative management, three aspects are covered (Baumert & Mark, 1992) Reuse support towards human resources

- Incentives and planning towards reuse
- Reuse level of previous projects ,applied to the strengthening of the level of improvement

The complete infrastructure graph recommended by (Prieto-diaz et. al, 1993) is shown in following figure.

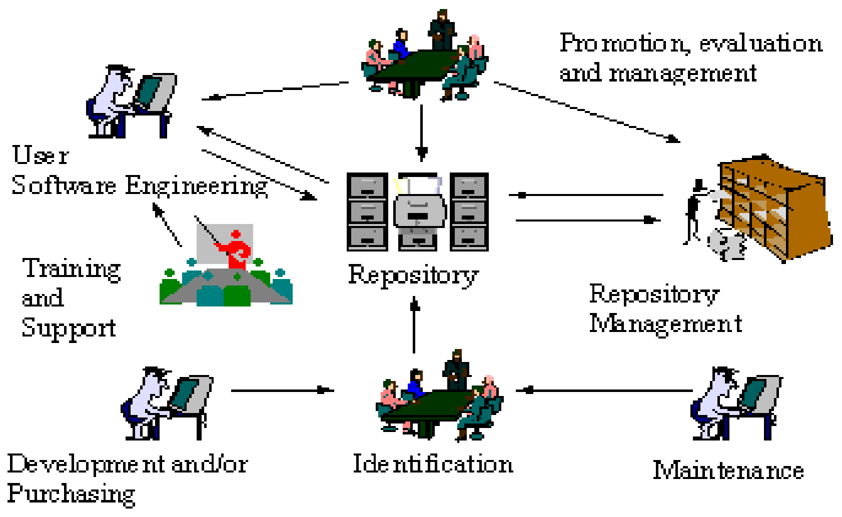

Fig. 7. Reuse Infrastructure

According to (Grady,1997), a reuse-driven process is a framework for performing domain-specific engineering which helps to optimize the software practices to build products of a particular type, resulting in improved productivity and product quality .This focusing and standardization of effort is the key to systematic reuse, leading ultimately to a manufacturing discipline.

As with any process, the CMM and RCM process improvement factors can motivate improvements in a reuse-driven process. In this, some of the corresponding RCM factors ought to be addressed in attaining CMM levels 2 and 3. The 17 process definition factors in the RCM concern differences in the types of reuse based process that an organization may adopt (Grady, 1997). The RCM, in its implementation model, defines four types of reuse-based process: opportunistic, integrated, leveraged, and anticipating. These types, ordered by increasing cost-risk and benefit, provided a categorization for the diversity of approaches already envisioned by the RSP methodology as a family of processes. These process definition factors do not fit into the proper scope of the CMM because they involve a choice among equally valid alternative process conceptions; no one approach is best for everyone.

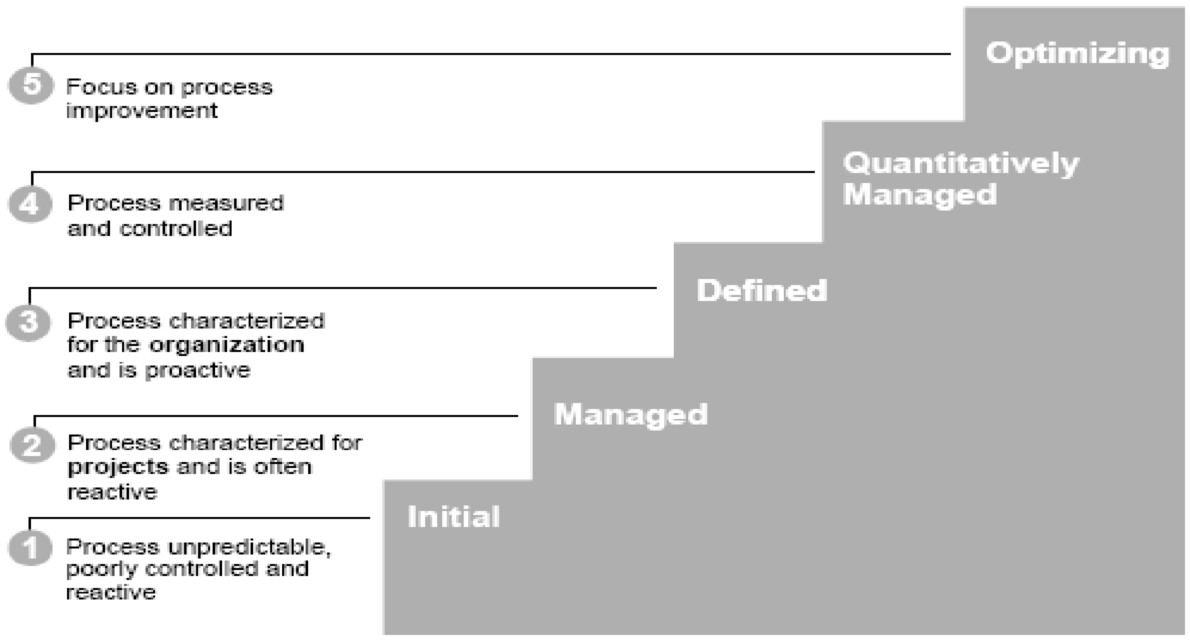
5 Focus on process improvement
4 Process measured and controlled
3 Process characterized for the organization and is proactive
2 Process characterized for projects and is often reactive
1 Process unpredictable, poorly controlled and reactive
Optimizing
Quantitatively Managed
Defined
Managed
Initial

Fig. 8. Traditional CMM Levels

6.6 Proposed CMM levels by the author for reuse

Although process maturity is extremely important in delivering high quality software, there is no standardized maturity model adopted for reuse based software development process. Some organizations which are primarily involved in reuse based software development are following some versions of reuse maturity models. After an extensive literature review (Almeida et al. ,2004; Koltun & Hudson,1991;Frakes and Fox ,1995; Rine, 1997a; Rine & Nada ,2000a) and from my study on reuse projects, I have identified some factors related to software reuse (Brito et al., 2006;Lucr´edio et al. ,2007), that were considered as a basis for this maturity model specification, in order to guide the organizations in the reuse evaluation and/or adoption.

RCMM is a maturity model with focus on reuse and describes which are basic in order to ensure a well planned and controlled reuse oriented software development. In RCMM, there are 5 levels inspired by SEI's Capability Maturity Model. Each level represents a stage in the evolution to a mature reuse process. A set of maturity goals for each level and the activities, task and responsibilities, needed to support are shown in the figure below.

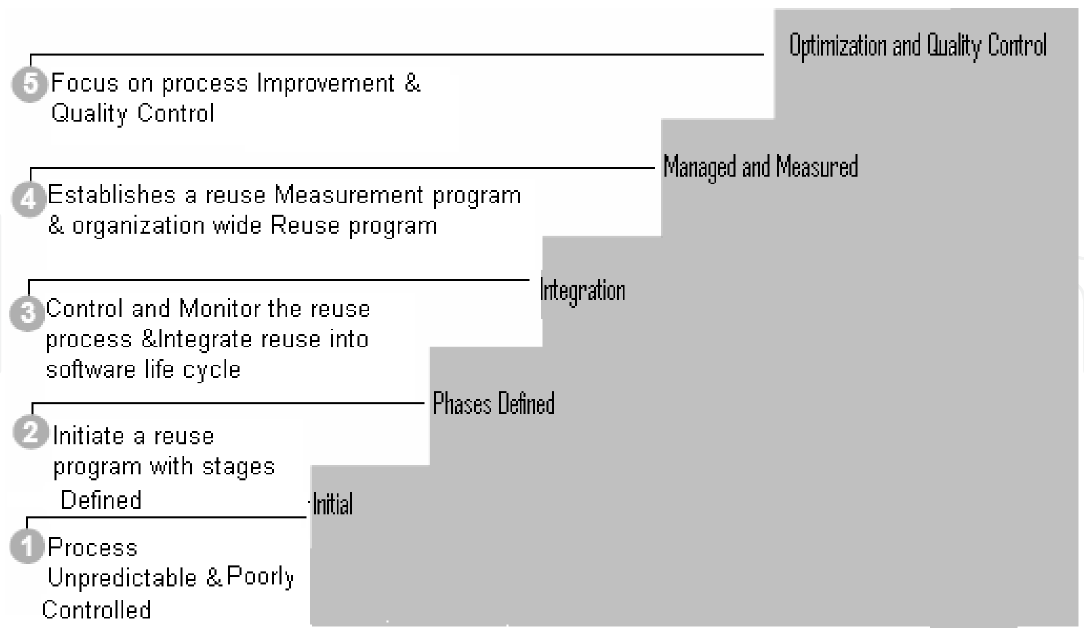
Optimization and Quality Control
5 Focus on process Improvement & Quality Control
Managed and Measured
4 Establishes a reuse Measurement program & organization wide Reuse program
Integration
3 Control and Monitor the reuse process &Integrate reuse into software life cycle
Phases Defined
2 Initiate a reuse program with stages Defined
Initial
1 Process Unpredictable & Poorly Controlled

Fig. 9. Proposed CMM levels by the author for reuse

The RCMM model suggested here can be used as s a basis for estimating the level of software reuse practice within an organization. As future work, Maturity Model aims at identifying the strengths of an organization with respect to software reuse and the opportunities for improvements can be adopted. Correct implementation of software reuse and the benefits for an organization adopting reuse in their processes can be evaluated only based on quantitative data. Therefore appropriate Reuse Business and Engineering metrics are recommended to be used within the maturity model to measure the achievement of the respective objectives, the efficiency of the applied practices and the quality of the results obtained. To evaluate the suggested model, it has to be put in the industrial environment and there is a need to get more feedbacks from experts to evaluate the current reuse practice stage and plan the next activities to implement the reuse program.

7. Summary

Good management can make a difference for success in the case of reuse based software development. Proper monitoring and control of the progress towards the business goals and the performance compliance needs an effective management program. However the nature of the reuse business changes the character and extent of the issues. By suitably applying some modifications to the traditional management techniques by keeping in mind the reuse business goals and proper planning will lead to success in reuse business.

By gathering issues associated with people, process and product measurements and by estimating and validating using estimation formulas and economic models ensures that proper usage of resources will follow the right process and right product.

Typically, significant cultural change is needed and will not happen overnight. Just-in-time education and ongoing senior management leadership are needed to make such cultural changes.

8. References

Abts, C., Boehm, B. & Bailey Clark, B., (2000a). COCOTS: a COTS software integration cost model, *Proceedings ESCOM-SCOPE 2000 Conference*, April.

Abts, C.; Boehm, B & Bailey Clark, B. (2000b). Empirical Observations on COTS Software Integration Effort Based on the Initial COCOTS Calibration Database, *ICSE 2000 COTS Workshop* - mip.sdu.dk

Almeida, E. S., Alvaro, A., Lucr´edio, D., Garcia, V. C., & Meira, S. R. L. (2004). Rise project: Towards a robust framework for software reuse. *IEEE International Conference on Information Reuse and Integration (IRI),* pp. 48–53, Las Vegas, USA.IEEE/CMS.

Balda, D.M. & Gustafson, D.A. (1990). Cost-estimation models for the reuse and prototype software development, *ACM SIGSOFT*, pp. 42-50, July.

Basili, V.R.; Briand, L. C. & Thomas, W. M. (1994). Domain Analysis for the Reuse of Software Development Experiences, *Proceedings of the 19th Annual Software Engineering Workshop*, NASA/Goddard Space Flight Center, November.

Balda, D. and Gustafson, D.(1990). Cost estimation Models for the Reuse and Prototype Software Development Life-cycles.*ACM SIGSOFT Software Engineering Notes* Vol. 15, No. 3, Pages 42-50, July.

Bass, L.; Clements & Cohen. S. et al. (1997). Product Line Practice Workshop Report, *Technical Report CMU/SEI-97-TR-003*, Software Engineering Institute, June.

Baumert, John; Mc Whinney &Mark.(1992).Software Measures and the Capability Maturity Model., Software Engineering Institute,*CMU/SEI-92-TR-25*, Pittsburgh, PA USA, September.

Bosch, J. & Molin, P. (1997). Software Architecture Design: Evaluation and Transformation, *Research Report* 14/97, University of Karlskrona/Ronneby, August.

Bosch, J. (1998a). Object Acquaintance Selection and Binding, *Theory and Practice of Object Systems*, February.

Bosch, J. (1998b). Product-Line Architectures in Industry: A Case Study, *Proceedings of the 21st International Conference on Software Engineering*, November.

Bosch, J. (1999). Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study. www:http://www.ide.hk-r.se/~bosch

Boehm, B.M & Papaccio, P.N.(1998).Understanding and controlling software costs, *IEEE transactions on software engineering*, 14(10), 1462-77

Boehm, B., Clark, B.et al.(1995) Cost Models for Future Software Lifecycle Processes: COCOMO 2.0, *Annals of Software Engineering*.

Boehm, B.W (1981). *Software Engineering Economics*, Englewood Cliffs, NJ Prentice Hall.

Boehm, B. & Abts, C.et al. (2000).*Software Cost Estimation with COCOMO II,* Prentice Hall, NJ, June.

Barnes, B.H. & Bollinger, T.B. (1991), Making reuse cost-effective, IEEE software, Vol. 8,No.1, pp. 13-24, January.

Banker, R. D. & Kauffman, R. J. et.al(1993).Evaluation of Software Reuse, *IEEE Transactions on Software Engineering*, Vol. 19, No. 4, pp. 379-389, April.

Brito, K. S., Alvaro, A., Lucr´edio, D., Almeida, E. S., and Meira, S. R. L. (2006). Software reuse: A brief overview of the brazilian industry's case. *5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE)*, Short Paper, Rio de Janeiro, Brazil. ACM Press.

Buschmann, F; Jäkel,C & Meunier,R.et al.(1996).Pattern-Oriented Software Architecture- A System of Patterns, John Wiley & Sons.

Chidamber, S. R. & Kemerer, C. F.(1994). A Metrics Suite for Object Oriented Design, *IEEE Transactions on Software Engineering*, vol. 20, pp. 476-493.

Dikel, D.; Kane, D. & Ornburn, S. et al (1997). Applying Software Product-Line Architecture,'IEEE Computer, pp. 49-55, August.

Frakes, W. & Terry, C. (1994).Reuse Level Metrics, *Proceedings of the 3rd International Conference on Software Reuse: Advances in Software Reusability, IEEE.*

Davis, T. (1993). The reuse capability model: A basis for improving an organization's reuse capability. *Proceedings of 2nd ACM/IEEE International Workshop on Software Reusability*, pp. 126–133. IEEE Computer Society Press / ACM Press.

Frakes, W. B. and Fox, C. J. (1995). Sixteen questions about software reuse. *Communications of the ACM, 38(6):75–87. ACM Press. New York, NY, USA.*

Frakes, W. & Terry, C. (1996).Software Reuse: Metrics and Models, *ACM Computing Surveys*, Vol. 28, No. 2, June.

Favaro, J. A. (1996).Comparison of Approaches to Reuse Investment Analysis. *Proceedings of the Fourth International Conference on Software Reuse, IEEE Computer Society Press*, pp.136-145, Los Alamitos, CA.

Geels. (2005). *Technological Transitions and System Innovations: A co-evolutionary and socio-technical analysis*, Edward Elgar, Cheltenham.

Guo, J. & Luqui.(2000).A Survey of Software Reuse Repositories, *7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, April.

Grady, Campbell.(1997) Tailoring Reuse-Driven Processes In A Process Improvement Context, *ERW-97 Position Paper*, October.

Graves, S. B.(1989).The Time-Cost Tradeoff in Research and Development: A Review, *Engineering Costs and Production Economics,* 16, pp. 1-9, Elsevier Science Publishers.

Griss, M. L., Favaro, *&* P. Walton.(1993). Managerial and Organizational Issues - Starting and Running a Software Reuse Program, *Software,* Shaefer, March.

Harris, K.(1992).Using an Economic Model to Tune Reuse Strategies", *Proceedings of the 5th Annual Workshop on Software Reuse.*

Henry, E. & Faller,B. (1995).Large-Scale Industrial Reuse to Reduce Cost end Cycle Time, *IEEE Software,* Vol. 12, No. 5, September.

Hafedh Mili; Fatma Mili & Ali Mili. (1995).Reusing Software: issues and research Directions, *Proc. IEEE trans. software Engineering*, Vol .21, No.6, June.

Henninger, S.(1997).An Evolutionary Approach to Constructing Effective Software Reuse Repositories, *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 2, pp. 111-140, April.

Henderson-Sellers, *B.(1996). Object-Oriented Metrics -Measures of Complexity. Upper Saddle River,* NJ, EUA: Prentice Hall PTR.

Heinemann*, G. T. &* Council, W. T.(*2001).Component-Based Software Engineering - Putting the Pieces Together*. Boston, MA: Addis on-Wesley.

Van Jacobson; Martin Griss & Patrik Jonsson. (2000).Software Reuse- Architecture, Process and Organization for Business Success, *ACM Press*.

Jacobson, I.; Griss, M. & Jönsson, P. (1997). Software Reuse - Architecture, Process and Organization forBusiness Success, Addison-Wesley.

Jasmine, K.S., & Vasantha, R.(2008a). Cost Estimation Model For Reuse Based Software products, Proc. *IAENG International MultiConference of Engineers and Computer Scientists 2008 (IMECS 2008),* Hong Kong, pp.951-954, March.

Jasmine, K.S., & Vasantha, R., (2008b). Effort Estimation in Reuse-Based Software Development Approach, *Proc.National Conference on Wireless network security –Issues & Challenges (WNSIC-08),* Department of CSE, R.V.College of Engineering, Bangalore, pp.63-67, March.

Johnson & Foote (1988).Designing Reusable Classes, *Journal of Object-Oriented Programming*,Vol. 1 (2), pp. 22-25.

Koltun, P. & Hudson, A., (1991). A reuse maturity model. *4th Annual Workshop on Software Reuse*, Hemdon, Virginia: Center for Innovative Technology.

Laraman, C. & Basili, V.R., (2003). Iterative and Incremental Development: A Brief History. *IEEE Computer*, 2003. 36(6): p. 47-56.

Larry McCarthy.(2002). Motorola CMMI Working Group,CMMI^{SM} Transition in a Commercial Environment, *2nd CMMI^{SM} Technology Conference and User Group*, Hyatt Regency Denver Technical Center, November, Denver, Colorado

Lim, W. (1994). Effects of Reuse on Quality, Productivity, and Economics, *IEEE Software*, Vol. 11, No. 5, September.

Llorens Morillo; A.Amescua Seco & Martinez Orga, V.,Carls III University of Madrid, Spain, *http://www.ie.inf.uc3m.es/grupo/Investigacion/LineasInvestigacion/Congresos/RMM97_Docum_Final.doc.*

Loingtier, Irwin, J. (1997).Aspect-Oriented Programming, *Proceedings of ECOOP'97,* pp. 220-242, LNCS 1241.

Lucr´edio, D., Brito, K. S., Alvaro, A., Garcia, V. C., Almeida, E. S., Fortes, R. P. M., and Meira, S. R. L. (2007). Software reuse: The brazilian industry scenario. *Journal of Systems and Software*, Elsevier.

Malan, R. &Wentzel, K. (1993).Economics of software reuse revisited, *Proc. 3rd Irvine Software Symposium*, University of California, Irvine, 30 April, pp.109-21.

Macala, R.R.; Stuckey, L. D. & Gross, D.C. (1996). Managing Domain-Specific Product-Line Development,*IEEE Software*, pp. 57-67.

Mascena, Almeida & Meira.(2005).A Comparative Study on Software Reuse Metrics and Economic Models from a Traceability Perspective, *IEEE*.

Paul Goodman. (1993).*Practical Implementation of Software Metrics*, McGraw Hill, London.

Poulin, J.S., & Caruso, J. (1993). A Reuse Metrics and Return on Investment Model, *Proceedings of the 2nd Workshop on Software Reuse: Advances in Software Reusability, IEEE.*

Poulin, J.S., (1994).Measuring Software Reusability, *Proceedings of the 3rd International Conference on Software Reuse: Advances in Software Reusability, IEEE.*

Poulin, J.S., (2006). The business case for software reuse: Reuse metrics, economic models, organizational issues, and case studies. *Tutorial notes.*

Prieto-D´ıaz, R. (1993). Status report: Software reusability. *IEEE Software*, 10(3):61–66, IEEE Computer Society Press. Los Alamitos, CA, USA.

Prieto-D´ıaz, R. & Frakes W.b.,eds (1993).Advances in software reuse: *Selected papers from the second International Workshop on software reusability*, Los Alamitos, California, March, IEEE Computer Society Press.

Pressman, R. (2001).Software Engineering-A practitioner's Approach, Fifth edition, McGraw-Hill.

Rotmans, Kemp & van Asselt. (2001) More Evolution than Revolution. Transition Management in Public Policy, Foresight 3(1): 15-31

Rine, D. C. (1997a). Success factors for software reuse those are applicable across domains and businesses. *ACM Symposium on Applied Computing*, pages 182–186, San Jose, California, USA. ACM Press.

Rine, D. C. and Nada, N. (2000a). An empirical study of a software reuse reference model. *Information and Software Technology*, 42(1):47–65.

Rine, D. C. and Nada, N. (2000b). Three empirical studies of software reuse reference model. *Software: Practice and Experience*, 30(6):685–722.

Rine, D. C. & Sonnemann, R. M. (1998). Investments in reusable software. a study of software reuse investment success factors. *The Journal of Systems and Software*, 41:17–32.

Rushby Craig& Bruce Allgood (2001).CMMI: A Comprehensive Overview.CMMI User Group, *Computer Resources Support Improvement Program*, Hill AFB, UT, November.

Sunita Chulani (2000).COCOMOII. *Wiley Software Engineering Encyclopedia*, Fall.

Svahnberg, M. & Bosch, J. (1999).Evolution in Software Product Lines: Two Cases., *Journal of Software Maintenance - Research and Practice*, 11(6), pp. 391-422.

Svahnberg, M. &Bosch, J. (1999) .Characterizing Evolution in Product-Line Architectures., *Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications*

Szyperski. C. (1997).*Component Software - Beyond Object-Oriented Programming*, Addison-Wesley.

Suz Garcia. (2003),On the TRIAL to CMMI® : A Framework for Effective Transition Management Practices, *SEI Technology Transition Practices*, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

WWW.sei.cmu.edu/publications/documents/*02.reports*/02tr007.html

SEI/CMU. (2003). INDIA 2003 presentation on TTP website www.sei.cmu.edu/ttp

SPC-CMC. (1993). Software Productivity Consortium, *Reuse-driven Software Processes Guidebook*, SPC-92019-CMC, version 2.0.

SPC-CMC. (1993).Software Productivity Consortium, *Reuse Adoption Guidebook*, SPC-92051-CMC, Version 2.0.

CMU/SEI. (1993). Software Engineering Institute, *Capability Maturity Model for Software*, CMU/SEI-93-TR-024, version 1.1.

Topaloglu,Y.;Dikenelli,O & Sengonca,H. (1996),Afour Dimensional reuse Maturity Model, Symposium on Computer and Information Sciences. ISCIS-XI, Antalya, Turkey. November.

http://en.wikipedia.org/wiki/COCOMO

http://sunset.usc.edu/COCOMO II/Cocomo.html

http://sunset.usc.edu/COCOTS/cocots.html.

Engineering the Computer Science and IT
Edited by Safeeullah Soomro

ISBN 978-953-307-012-4
Hard cover, 506 pages
Publisher InTech
Published online 01, October, 2009
Published in print edition October, 2009

It has been many decades, since Computer Science has been able to achieve tremendous recognition and has been applied in various fields, mainly computer programming and software engineering. Many efforts have been taken to improve knowledge of researchers, educationists and others in the field of computer science and engineering. This book provides a further insight in this direction. It provides innovative ideas in the field of computer science and engineering with a view to face new challenges of the current and future centuries. This book comprises of 25 chapters focusing on the basic and applied research in the field of computer science and information technology. It increases knowledge in the topics such as web programming, logic programming, software debugging, real-time systems, statistical modeling, networking, program analysis, mathematical models and natural language processing.