# DEVELOPING REUSABLE SOFTWARE COMPONENTS FOR DISTRIBUTED EMBEDDED SYSTEMS

**K. Subba Rao[1], S. Naga Mani[2], M.Santhosi[3], L. S. S. Reddy [4]**

*[1]Associate Professor, IT, LBRCE, Mylavaram, India,* **ksubbarao_22@yahoo.co.in**
*[2]Assistant Professor, IT, LBRCE, Mylavaram, India,***luckiestmani@gmail.com**
*[3]Assistant Professor, CSE, LBRCE, Mylavaram, India,* **santhosi550@gmail.com**
*[4]Professor& Director, CSE, LBRCE, Mylavaram, Indi a,* **director@lbrce.ac.in**

## Abstract

*Software reuse is one of the technical approach that many believe can reduce software development time and cost. Reuse is clearly a partial solution to the long and costly development problems with complex control systems.This paper discusses why software is hard to reuse and why we cannot extend reuse analogies in other fields to software. The proposed approach is based on an incremental strategy and addresses managerial, economic, performance and technology transfer issues. This approach is practical, effective, and has potential to make reuse a regular practice in the software development process. This paper explores what is necessary for accomplishing systematic reuse and recommends strategic approach for reuse of software*

***Index Terms:*** *Software reuse, Systematic reuse, Software development, Incremental strategy*

---------------------------------------------------------------------***---------------------------------------------------------------------

## 1. INTRODUCTION

The concept of reuse in engineering is simple. It has been successfully applied in several areas, from civil engineering to electronics. In fact most engineering disciplines are based on the reuse concept from components to formulas to ideas. Despite several years of trying to bring reuse to practice, software engineers have found out that reuse in software is not the same as in other areas that software is very hard to reuse.

### 1.1 Reuse in Conventional Engineering

Reuse in traditional engineering is part of the engineering process. In aeronautical design, for example, engineers are thought that airfoils exhibit certain performance characteristics based on their geometry and that lift to drag ratio is an important factor of an airfoil. When designing a wing, an airfoil is usually selected from a catalog of tested and certified airfoils (e.g., NACA). An airfoil defines the cross section geometry of a wing and defines most of the wing characteristics. The selection is based on the wing performance requirements. Practice in making the choice that provides the best balance in the requirements and the one that does not compromise key requirements is what makes a good engineer. Engineers are, therefore, good reusers by training, even if they never took a reuse course.

### 1.2 Reuse in Software Engineering

The state of the practice in software engineering has been, by far, to blindly satisfy the given requirements. Significant work has been done on requirements verification, that is, to ascertain that a final design or product truly satisfies the given requirements. This implies that once a set of requirements are given, and accepted, they are never questioned. The goal of the software engineer is to satisfy them, completely and correctly. Unfortunately, software engineers are not trained like other conventional engineers—to make optimum trade-off decisions. Implicit reuse is not part of their regular practice; reuse is explicit and has to be acquired later. In fact, except for modern software engineering courses, software engineering students have been typically trained for individual work where they are asked to create their own programs from scratch.

### 1.3 Developing Software Hard

It seems that software's main advantage-its softness-is suddenly becoming its major liability. As long as we see software as a malleable object, agreeing on conventions and standards will be a continuous uphill battle. How can we inject some element of hardness into software to make it more amenable to standardization? One answer is complexity. By making software elements more complex it becomes less attractive to create them and more attractive to reuse them. Defining sets of standard domain specific software components could be a relatively straight forward process. Committees of domain experts could analyze their own domains, identify common functions, engineer them, and declare them the standard components. This idealized standardization is in sharp contrast with reality. In the world of software it is the market who defines the standards, and those standards are not necessary the best (e.g., DOS, Windows.) Standards evolve, for the most part, not by imposing them (e.g., Ada) but by free choice (e.g., C, Unix.). Visual Basic and its exploding industry of VB ad-ons is a clear example of both arguments: complexity stimulates reuse, and market trends define standards. Making software components harder to create and easier to buy will stimulate software engineers to become true designers in the traditional engineering sense—to practice genuine trade-off analysis. Applications involving Visual Basic, for example, will call for use of commercially available components. Such applications will exhibit, with time, certain common features that will be adopted as standards. The role of the software engineer will resemble more closely the role of a traditional engineer: selecting the best combination of existing components that will generate a "good," not necessarily the "best," design. And one that will force some compromises on the requirements.

## 2. RELATED WORKS

The proposed an incremental, systematic, and formal approach to establish reuse programs in software organizations. A reuse program is an organizational structure and collection of support tools aimed at fostering, managing, and maintaining the practice of reusing software in an organization. The strategy in incremental approach is that provides continuous feedback for correction and adaptation allowing for later integration of critical issues left aside in the beginning.

### 2.1 Reuse in Conventional Engineering

Embedded systems have managed to spread rapidly over the past few decades to be virtually in any kind of today appliances such as digital watches, set-top boxes, mp3players, washing-machines, mobile telephones, cars, aircrafts, forest machines and many more. Also, many embedded systems have to observe real-time constraints which mean that they must react correctly to events in an appropriate amount of time, neither too fast nor too slow. When all the timing requirements must strictly be ensured, embedded systems are called hard real-time systems whereas soft real-time systems are more flexible in that sense that they can tolerate to miss some timing requirements without generating negative effects. The respect of timing constraints is of prime importance for maintaining the safety of the physical device which relies on it. One major issue in dealing with safety-critical real-time embedded system is therefore to ensure that the system behaves correctly even in the worst possible situations.

### 2.2 Distributed Embedded Systems

Distributed embedded systems developed by several organisations are common in, e.g., the vehicle industry and in the automation industry. The systems consist of several computer nodes connected with one or several networks, where each node can be developed by different organisations specialised on different areas. For example, a modern car in the premium segment has 40 or more computers (Electronic Control Units (ECUs)), where the engine control ECU comes from a specialized engine developing organization, and the climate control ECU is developed by an organisation that focuses on the passenger cabin. These different ECUs can be seen as large and complex COTS components, they contain hardware (processor, communication hardware, memories, and I/O units), and software (operating system, device drivers, and control system software).

### 2.3 Why Is Reuse Not Delivering

Software reuse is still far from crystallizing the ideas of a software industry based on interchangeable standard parts. The problem is not lack of tools and technology but unwillingness to address managerial, economic and several other issues influencing software reuse. We find ourselves with a full toolbox but unable to use these tools effectively. There is a need to learn what the relationships are between the technical and nontechnical aspects, how one influences the other, and what makes a reuse program a success or a failure.

### 2.4 Model for Implementing Reuse Programs

This paper proposes an incremental, systematic, and formal approach to establish reuse programs in software organizations. A reuse program is an organizational structure and collection of support tools aimed at fostering, managing, and maintaining the practice of reusing software in an organization. The remaining of the paper presents a model for creating and operating reuse programs in an organization and makes reference to some successful reuse experiences that support the ideas of this model. Figure 1 is a high level view strategy for implementing reuse programs. A key ingredient is management support which is a common factor in all successful reuse programs Management commitment is essential because reuse programs demand changes in the way software is developed. Current methodologies and procedures do not include reuse as part of their process. Management must provide the necessary company resources required to start, evolve, and operate a reuse program and also make available necessary key information for assessing the reuse potential of the organization. The outcome of the "assess reuse potential" activity is an assessment report that includes an implementation plan. The implementation plan controls the reuse program. The inputs to the reuse program (i.e., "implement reuse program incrementally") include software from existing systems and requirements for future systems. The products of a reuse program include a series of software catalogs, an automated library system, generic architectures, and a collection of reusable components.

### 2.4 Assessment Report

The proposed assessment report includes: feasibility analysis, domain suitability assessment, cost-benefit analysis, and an implementation plan. Below are some of the questions that should be answered by each document.

*Feasibility analysis*- Does the organization have enough resources (financial and human) to implement a reuse program? Can the organization afford it? Is reuse necessary in the organization? Does the organization want to do it? Is management alternative? (compare and contrast). Does a critical mass of software engineers exist? Is software production large enough to justify a reuse program? What is the software volume produced by

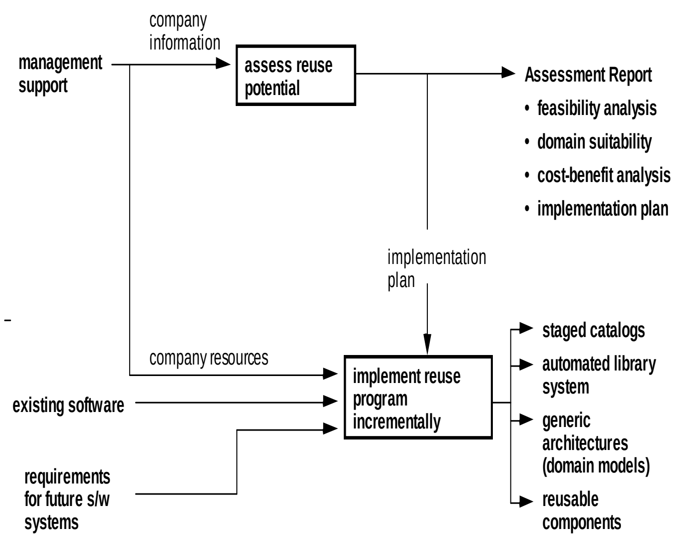

**Figure 1-** Strategy for Implementing Incremental Reuse Programs

- *Cost benefits analysis*- How much does it cost? (cost schedule). Is a reuse program economically feasible? Is it worth doing it? What alternatives exist for implementing a reuse program? What is the scope? How big a program is it contemplated? (corporate level, division, project, etc.). What are the expectations? What is the desired level of reuse? (partial, opportunistic, formal, total).
- *Implementation plan*- The incremental model below explains the implementation details. The model can be used to provide time and cost schedules.

## 2.5 Implementing Reuse Program

Figure 2 shows the proposed model. A reuse program can be implemented in four basic stages: Initiation, Expansion, Contraction, and Steady State.

- *Stage 1: Initiation*- Existing software is analyzed to select potentially reusable components. Component descriptors are extracted manually or automatically and a preliminary index is produced. The index can be generated using free text information retrieval techniques based on word frequency analysis [Salt83]. A stage 1 catalog is generated and distributed. This first catalog informs software engineers in the organization about potentially reusable software. Stage 1 can be started at a project level involving one part-time individual at almost negligible cost. This first catalog raises the level of awareness about reuse in the project and stimulates other individuals to identify potentially reusable components.
- *Stage 2: Expansion*- The size of the catalog increases as more of the existing software is identified for reuse, thus, placing a demand for a classification scheme. An initial faceted classification scheme is produced and included with the stage 2 catalog. Based on data from the feasibility study, an automated library system could be contemplated for supporting distribution and availability of the catalog, and for refinement and maintenance of the faceted classification. A faceted scheme also provides basic domain models in the form of taxonomies and standard descriptors or lexicons which in turn support bootstrapping the domain analysis process. This second stage requires more resources than stage 1. A part time domain expert and a part time librarian are sufficient for a reuse program involving two or three projects. Further growth is stimulated by the availability of an automated, well organized catalog.
- *Stage 3: Contraction*- Domain analysis is the key activity in this stage. Early domain models from stage 2 together with more detailed information from existing systems and from requirements for future systems are used for domain analysis. Standard architectures and functional models are derived and common components are grouped to support basic generic functions. Redundant and ineffective components are identified and retired from the collection leaving only components that support functions and features of the domain architecture. This results in a contraction in the size of the collection. The collection and classification are updated and a stage 3 catalog made available. Stage 3 requires the most resources.

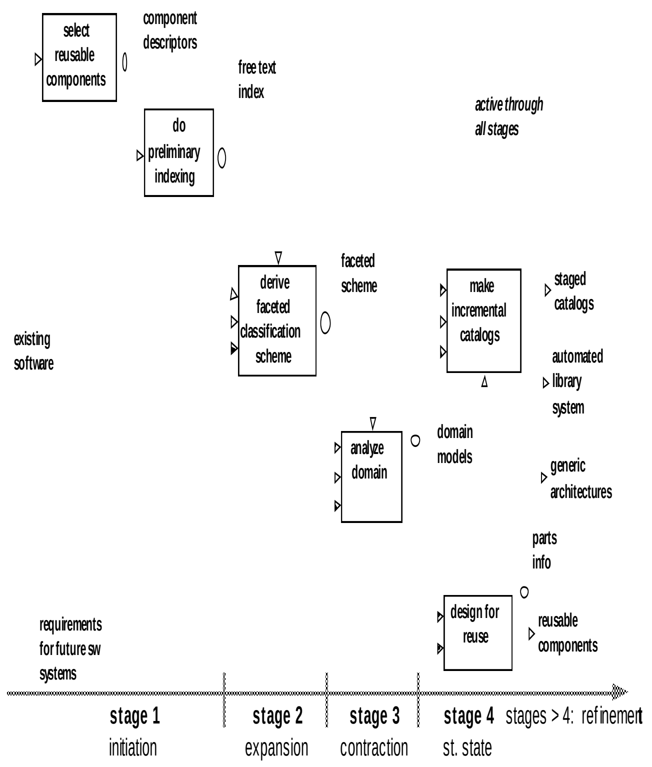

**Figure 2-** Reuse Program Implementation

- *Stage 4: Steady State*- After identifying the essential elements for software systems in a specific domain, existing components are progressively replaced by components supporting domain specific functions. These components are explicitly designed to be reusable since they plug directly into the domain architecture. Further stages of the program should not increase the size of the collection but only make current components more efficient and reliable. Performance and reusability data is fed-back to domain analysis for model refinements. New catalog editions are made available periodically. The required resources for a program involving several projects may include a domain analyst, several domain experts and two or more software engineers (also called domain engineers).

## 2.5 Implementing Reuse Program

The proposed model for implementing reuse programs proceeds incrementally. It is also systematic and formal. Incremental means the program is implemented in progressive steps or stages where each stage sets the basis for the following stage. Some advantages include:

- Provides immediate return on investment.
- Builds confidence within the organization.
- Easier to manage.
- Allows for tuning and refining the reuse process.
- Facilitates monitoring and evaluating reuse.

Systematic means that the process is consistent and repeatable and follows a logical progression of events. The following are some of the advantages:

- Makes reuse an integral part of software development.
- Makes reuse a standard practice with a potential to become compulsory.
- Helps towards a better software development methodology.
- Makes everybody a participant.
- Promotes a reuse culture.

Formal means the process can be decomposed into well defined steps, each being complete and described in some accepted representation. Advantages of formalizing reuse include:

- Promotes creation of standards
- Improves quality and reliability
- Facilitates management control
- Helps in identifying support tools
- Increases potential for reuse across organizations

## 2.6 Observations

The proposed model provides a learning environment to enable domain analysis and a basic mechanism to establish reuse programs. A library system, for example, should be seen as an instrument to achieve reuse not as the objective of reuse as is often the case. The incremental nature of the model provides the needed integration of people, tools, and processes. It also encourages management to support and participate in it. The main economic advantage of this approach is an immediate return on investment. Organizations have avoided implementing reuse programs because of the perceived need for a large initial investment, which management may find difficult to justify, and because of the uncertainty about the success of reuse. The strategy in this model is to start with a very small initial investment and to justify each further step with the results of the previous step.

## 2.7 Required Organizational Structure

An organizational structure is essential to establish a successful reuse program. A basic structure includes six groups:

- *Asset Management Group*- provides initiatives, funding, and policies for reuse.
- *Identification and Qualification Group*- identifies potential reusability areas and collects, procures, and certifies new additions to the collection.
- *Maintenance Group*- maintains and updates reusable software components.
- *Development Group*- creates new reusable components

- *Reuser Support Group*- assists and trains users and runs tests and evaluations of reusable components.
- *Librarian*- updates and distributes catalogs, classifies new assets, maintains library system, and manages asset orders.

This list defines the basic roles in a reuse program. It does not imply that each role is assigned to one or more individuals. During Stage 1, for example, all six roles may be assigned to one individual, but, as the program evolves, selected in-dividuals may be assigned to specialized roles. A corporate wide program during contraction mode, for example, may have a staff of 10 or more.

### 2.8 Required Organizational Structure

- *Stable*- the same structure supports all stages of a reuse program
- *Flexible*- roles and people can be changed without affecting function
- *Evolvable*- may start with a minimum of one person in multiple roles and evolve to multiple teams with specific roles.
- *Practical*- provides an infrastructure for the practice of reuse.
- *Effective*- focused tasks by defining specific roles.
- *Economical*- cost, complexity, and size adjustable to organization budget.

## CONCLUSIONS

A model for implementing reuse programs has been presented. The model is procedural and defines activities essential to enabling reuse programs in software organizations, and addresses several of the factors that impede the effective practice of reuse. The model is incremental, sys-tematic, and formal and is based on evidential experience.

## REFERENCES:

[1]. Lanergan, R.G. and B.A. Poynton "Reusable Code: The Application Development Technique of the Future." In Proceedings of the IBM SHARE/GUIDE Software Symposium, IBM, Monterey, CA, October, 1979

[2]. Matsumoto, M. "SEA/I: Systems Engineer's Arms for Industrialized Production and Support of Application Programs". In Proceedings of 6th International Conference on Software Engineering, pp 39-40, Tokyo, September, 1982.

[3]. McIlroy, M.D. "Mass-produced Software Components". In Software Eng. Concepts and Techniques, 1968 NATO Conf. Software Eng., ed. J.M. Buxton, P. Naur, and B. Randell, pp 88-98, 1976.

[4]. Prieto-Díaz, R. "Implementing Faceted Classification for Software Reuse". Communications of the ACM, (April, 1991).

[5]. Salton, G. and M.J. McGill, Introduction to Modern Information Retrieval. McGraw-Hill, New York, 1983

[6]. Swanson, M.E. and S.K. Curry, "Implementing an Asset Management Program at GTE Data Services". Information and Management 16, 1989.

[7]. Sandeep Agrawal, Pankaj Bhatt. Real-time Embedded Software Systems An Introduction, August 2001.

[8]. Xia Cai, Michael R. Lyu, and Kam-Fai Wong. Component-Based Embedded Software Engineering: Development Framework, Quality Assurance and A Generic Assessment Environment, International Journal of Software Engineering and Knowledge Engineering Vol. 12, No. 2 (2002) 107.

[9]. J.M. Boyle and M.N. Muralidharan, “Program Reusability through Program Transformation,” IEEE Transactions on Software Engineering, vol. SE-10, no. 5, September 1984, pp. 574-588.

[10]. D'Alessandro, M. Iachini, P.L. Martelli, “A the generic reusable component: an approach to reuse hierarchical OO designs” appears in: software reusability, 1993.

[11]. M. BjÄork. QoS management in configurable real-time databases. Master's thesis, Department of Computer Science, LinkÄoping University, Sweden, 2004.