Malaysian Journal of Computer Science, Vol. 12 No. 1, June 1999, pp. 67-75

HYBRID CONVERGENCE LIFE-CYCLE MODEL FOR LARGE SCALE PROJECTS

Vasuthevan Balakrishnan
Monash University
Sunway Campus
No. 2, Jalan Kolej, Bandar Sunway
46150 Petaling Jaya
Selangor Darul Ehsan
Malaysia
email: vasu@academic.musm.edu.my

Sai Peck Lee
Faculty of Computer Science and
Information Technology
University of Malaya
50603 Kuala Lumpur
Fax : 603-7579249
email: saipeck@fsktm.um.edu.my

ABSTRACT

Much has been said about the problems that software engineers face in developing large systems. Traditionally, developers have adopted structured analysis and design methodology in developing systems that has been working fairly efficiently until now. Much complaint has been thrown at this methodology. As with this methodology, most complex software systems raise numerous software engineering problems such as design, development, requirements compliance, evolution and maintenance. To these problems, Object-Oriented (OO) methods and tools claim to bring a solution. The trend now is to use OO technology. But is this new approach mature enough for developing large and industrial strength applications? This paper presents an overview of the two methodologies, assesses its strength and weaknesses and suggests a new model for efficient development of large-scale systems incorporating both the strengths of structured and object-oriented methodologies.

Keywords: ***Large Scale Systems, Structured Analysis and design methodology, Object-Oriented technology***

1.0 INTRODUCTION

Systems designers are becoming more and more interested in object technologies, not only because of the recent explosion of methods and tools and the hype about them, but also because of the technological revolution they seem to imply. These potential users see in them a paradigm closer to the real world, an abstraction capability higher than traditional approaches, and above all, the long dreamed of benefits of reusing components and of a modularity allowing an easy evolution of systems [1].

But what are the real consequences, advantages and drawbacks of those technologies on a large-scale system, which is equivalent to hundreds of thousands of traditional lines of code? What about those organizations that have systems developed using traditional systems development life cycle that are working perfectly now and wish to incorporate OO technology in these systems. Will this be impossible with the constraints of budget, time, and market share?

This paper reviews the strengths and weaknesses of both ***structured*** and ***object-oriented*** methodologies and proposes a new life-cycle paradigm that addresses these issues. This life-cycle model, referred to as the hybrid convergence model, filters the weaknesses of both methodologies and incorporates their strengths. The model derives its strength from a number of features, which most software engineers would agree as necessary prerequisites for successful systems development. These features are: faster software production, reengineering of legacy systems, reusability of components, concurrent system development, consistent user involvement in the development of systems, and a continuous process of refinement through validation and verification up to the point of systems implementation. In order to achieve the required model, a review of the matter is presented. This comparative review may be used to justify the real needs for the development of a software development process model.

2.0 CONVENTIONAL VS OO APPROACH

Is OO Analysis (OOA) really different from the structured analysis approach? Structured analysis takes a distinct input-process-output view of requirements. Data are considered separately from the processes that transform the data. System behavior, although important, tends to play a secondary role in structured analysis. The structured analysis approach makes heavy use of functional decomposition. Fishman and Kemerer [1] suggest eleven "modeling dimensions" given below that may be used to compare various conventional and OOA methods.

1. identification/classification of entities
2. general to specific and whole to part entity relationships
3. other entity relationships
4. description of attributes of entities
5. large scale model partitioning
6. states and transition between states
7. detailed specification of functions
8. top-down decomposition
9. end-to-end processing sequences

10. identification of exclusive services
11. entity communication (via messages or events).

As there are many variations that exist for structured analysis and dozens of OOA methods are available, it is difficult to develop a generalized comparison between the two methods. It can be stated, however, that modeling dimensions 8 and 9 are always present with structured analysis and are rarely present when OOA is used [2].

2.1 Strengths of OO Methodologies

Object technologies do lead to a number of inherent benefits that provide advantages at both the management and technical level. These are:-

1. Correctness – The reuse of existing software components of OO methodologies allows prototyping systems to be built before the actual systems are developed. Here it differs from prototyping in structured methodology by the fact that the the class itself is closer to the users view of the real world objects and hence even before implementation, the users could validate the accuracy of the model. In structured methodology, prototyping is mainly done after the code implementation, which is actually an implementation model.

2. Robustness – There are two approaches of handling abnormal conditions. The first approach is to ignore the errors, change the system to a known ‘safe’ state, and continuing to run in some ‘degraded’ manner. OO methodologies support this approach by keeping the ‘safe’ state values in the ‘safe’ state variables within each object and copying them back when abnormal conditions are detected. Overheads are reduced because only the state values need to be restored. The second approach is to restore the system to the pre-erroneous condition. OO methodologies support this approach by keeping a copy of all the state values and restoring them on detection of errors.

3. Extendibility – Using inheritance, new objects can be built out of old ones. Systems can be extended quickly using this concept.

4. Maintainability – The modularity, inherent structure, and the insulation of objects from one another help to facilitate system maintenance. Changes can be made within one class without affecting others.

5. Reusability – Reusability of existing software components can be promoted in two ways. Firstly, new classes can be built out of old classes through inheritance. This reduces the need to build new classes from scratch. Secondly, systems can be built by using existing libraries either directly or with some modifications.

6. Integrity – ability of protecting data from corruption is achieved through encapsulation and information hiding. Only the encapsulated operations within an object can access the data structures of that object. This means that the data structures of an object are hidden from another object. Information hiding protects data from invalid access and thus from corruption.

7. Increased Modeling power - The increase in modeling flexibility and modeling power enables more complex systems to be built. Most of the system problems are detected in the early stage of design. Designers first identify the important system components (objects) and their relationships without having to worry about the implementation details that may complicate the overall design of the system.

8. Smooth transformation – OO systems closely model real - world systems. There is also a smooth transition from analysis to design because of the same concepts used throughout these phases [3].

2.2 Weaknesses of OO Methodologies

Object methodology is not without its failings. Some of its weaknesses are being addressed but at a rather slow pace. The weaknesses in object methodology are not mainly technical but its slow absorption by users in applying the methodology into existing systems development. Some of the weaknesses are:-

1. Lack of system decomposition – Decomposition is important because many systems are too large to be developed by one team in order to meet the deadline. Therefore, systems need to be broken down into smaller components that can be assigned to multiple teams to be worked on concurrently. Decomposition needs to be initiated at an early stage of the development process. Decomposition must have a strong connection between components to allow smooth integration of the components at a later stage. In OO methodologies there are difficulties in breaking systems down into smaller components.

2. Lack of end-to-end process modeling – According to Fichman and Kemerer (1992), global processes, which involve forward and backward execution of intermediate steps between start and end, exist in many problem domains. Although OO methodologies use operations to model parts of the process, there is no specific model to describe global processes. Therefore, a separate tool is required to arrange different encapsulated operations into a model that describes global processes.

3. Lack of supporting programming languages /programming skills in OO – Although OO methodologies can be implemented using traditional

programming languages, special languages supporting OO features are needed to facilitate the implementation of the OO systems. OO programming languages must have the ability to create classes, objects, subclasses {through inheritance), and to support messaging and dynamic binding. Only recently that programming languages such as JAVA and C++ are beginning to be widely used by organizations especially with the proliferation of the Internet. Many organizations cannot benefit from the use of OO methodologies until there are sufficient trained programmers who are skillful in OO programming and methodology.

4. Lack of supporting databases and tools – OO databases are required to store objects permanently on files. Currently, there are not many data base management systems that can handle applications such as CAD/CAM, which require many complex objects to be created and stored. Quality OO tools is needed to automate the analysis and design processes. Currently there are some OODBMS such as GEMSTORE, O2, JASMINE, etc., but user awareness or extensive usage is still limited.

5. Lack of reusable software – Software reusability claimed by OO methodologies may be difficult to implement and achieve. Pre-defined objects need to be well catalogued, documented, and easy to understand to facilitate their reuse. DCOM and CORBA are two examples of OO software architecture that addresses reusable component issues, but again the application is not widespread. Companies have to make a strong commitment to change to these new methodologies in order for them to be implemented. Investment at the front end must be made in order to reap the benefits from this new approach. Software components must be carefully designed to allow future reusability. The whole organization must be involved in identifying and designing organization-wide objects which help to establish the basis for stable, long term systems. Increased training costs are required for this approach.

2.3 Weaknesses of Structured Methodology

Why is there a change of mindset in the use of structured methodology that has stood the test of time? Is it because software professionals simple yearned for a "new" approach, or is it because systems are becoming too large and complex due to the proliferation of the internet, intranet, distributed computing and concurrent processing which structured methodology failed to address? Most software engineers would agree that the following could be the reasons why structured methodology is being replaced by object methodology.

1. Difficulties in managing more complex systems- Planning and control become increasingly difficult as the software systems become more complex. As a result, many systems are overdue, over budget, and of little use and limited quality. Many development projects even grow out of control and eventually have to be terminated.

2. Difficulties in handling unstructured problem – Existing structured methodologies are designed mainly to handle structured problems where system behavior is quite well understood and may be easily described using processing algorithms. However, they are inadequate in handling complex, unstructured problems. The focus of structured methodologies is in the solution to the problem rather than in the problem itself. There can be a huge gap between the actual problem and its representation and implementation. Traditional languages are not designed to handle complex relationships and natural structure. Problems modeled using conventional methodologies can easily display characteristics very different from the actual situation.

3. Low software reusability – Development productivity is low resulting from low reusable of past or present software components. Many functional equivalent modules that already exist are redeveloped again because either their existence is unknown or they are difficult to understand. There have been many unsuccessful attempts to construct reusable software component libraries. Only a few organizations have been successful at reusing software in a systematic way and on a large scale.

4. Poor software design techniques – Programs and data are linked together in complex and subtle manners. It is often difficult to understand and identify interactions between software components. Common data is not protected and can easily be misused and corrupted, leading to low system quality. Users are responsible for checking for data integrity, binding and correct data typing. Applications need to be programmed so that the user can properly understand and use common data structures.

5. Difficult and costly systems maintenance – A large portion of the software costs over its lifecycle is spent on maintenance. Maintenance is an important activity because systems are changing as requirements are changed, enhancements are made, problems are corrected, and operating environments are updated. Historically, system development does not include consideration for maintenance and enhancement. Therefore, developers often have to work with poorly written documentation and ill-structured code.

2.4 Strengths of Structured Methodology

Structured methodology not only derives its strength from its separation of process and data in its analysis model but also, due to its "divide and conquer" approach that most

developers find easy to understand. Some of its strengths are:-

1. In structured analysis, systems are analyzed by processes and by data within these processes i.e. using the dataflow diagram, entity relationship diagrams, and functional decomposition diagram. The advantage of analyzing systems by separating data and processes is that in very large systems, there could be myriad functions and data that has to be analyzed to avoid redundancy and duplication of data and functions. Functional decomposition helps the systems engineers to break-up a complex function into its subfunctions and subfunctions into their tasks.

2. In structured methodology, the completion of one phase triggers the start of another phase e.g. systems design starts only after when systems analysis has been completed. This ensures that the preceding phase must be thoroughly verified and completed so that errors, misrepresentations, and ambiguities does not get carried on to the next stage. The successful development of an information system requires that we follow the SDLC phases in order; that is, we must complete one phase one phase before we start on the next.

3. Focus on end products – Each phase of the Systems Development Life Cycle (SDLC) culminates in end products. Each end product represents a milestone or checkpoint in the information system's development and signals the completion of a specific phase. Management uses each checkpoint to assess where the development stands and where it should go next. Management's choices at each checkpoint are to proceed to a subsequent phase, to redo portions of the work just completed, to return to an earlier phase, or to terminate the development entirely. One major factor in management's decision is the quality of the end product. Because the end product from the SDLC phases is highly visible measures of the developers progress, it is imperative that attention be focused on the content and quality of these end products.

3.0 ANALYSIS PRINCIPLES

Over the past two decades, investigators have identified analysis problems and their causes, and have developed a variety of modeling notations and corresponding sets of heuristics to overcome them. Each analysis method has a unique point of view. However, all analysis methods are related by a set of operational principles:

1. The information domain of a problem must be represented and understood
2. The functions that the software is to perform must be defined
3. The behavior of the software (as a consequence of external events) must be represented
4. The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion
5. The analysis process should move from essential information toward implementation detail.

By applying these principles, the analyst approaches a problem systematically. The information domain is examined so that the characteristics of function and behavior can be communicated in a compact fashion. Partitioning is applied to reduce complexity. Essential and implementation views of the software are necessary to accommodate the logical constraints imposed by processing requirements and the physical constraints imposed by other system elements [2].

Just like structured systems analysis, OO approach must follow a development cycle. Many of the early object modeling methods concentrate on the data representations and usually applied to only a subset of the development process. Thus, for example, Coad & Yourdon (1990) concentrated on using objects to develop analysis models, whereas Booch (1993) primarily concentrated on the design model. A methodology requires models at each development phase with techniques to convert models developed at one phase to those at the next phase. The analysis principles described above are used to derive the proposed model.

4.0 HYBRID CONVERGENCE MODEL FOR LARGE SCALE SYSTEM

One of the most important objectives of information systems development is to deliver high quality software. This implies that the delivered software must satisfy both user and business requirements, be functionally operational and be adaptable to environmental change. Much of the research work carried out within information systems in the past three decades has been to improve the quality of the delivered software, through quality development process, i.e. through a complete and refined life-cycle model. The hybrid model that we propose in this paper can be used for evolutionary development as well as for synthesis of new systems. As shown in Fig. 1 and Fig. 2, this model addresses post-implementation problems at a very early stage of the development cycle such as maintenance, which can be time-consuming and costly, through a process of prototyping and refinement.

In the hybrid model, the process of analysis, design, and testing goes on in a cycle through prototyping, validation and verification with heavy user involvement. Here we start with some functions, decompose them into classes and then objects, connect them into a system, experiment with the system and so on. We can begin to add new functionality to design by allowing objects to be easily put together to form systems. Alternatively, we can begin to specialize objects and store them into a library and then

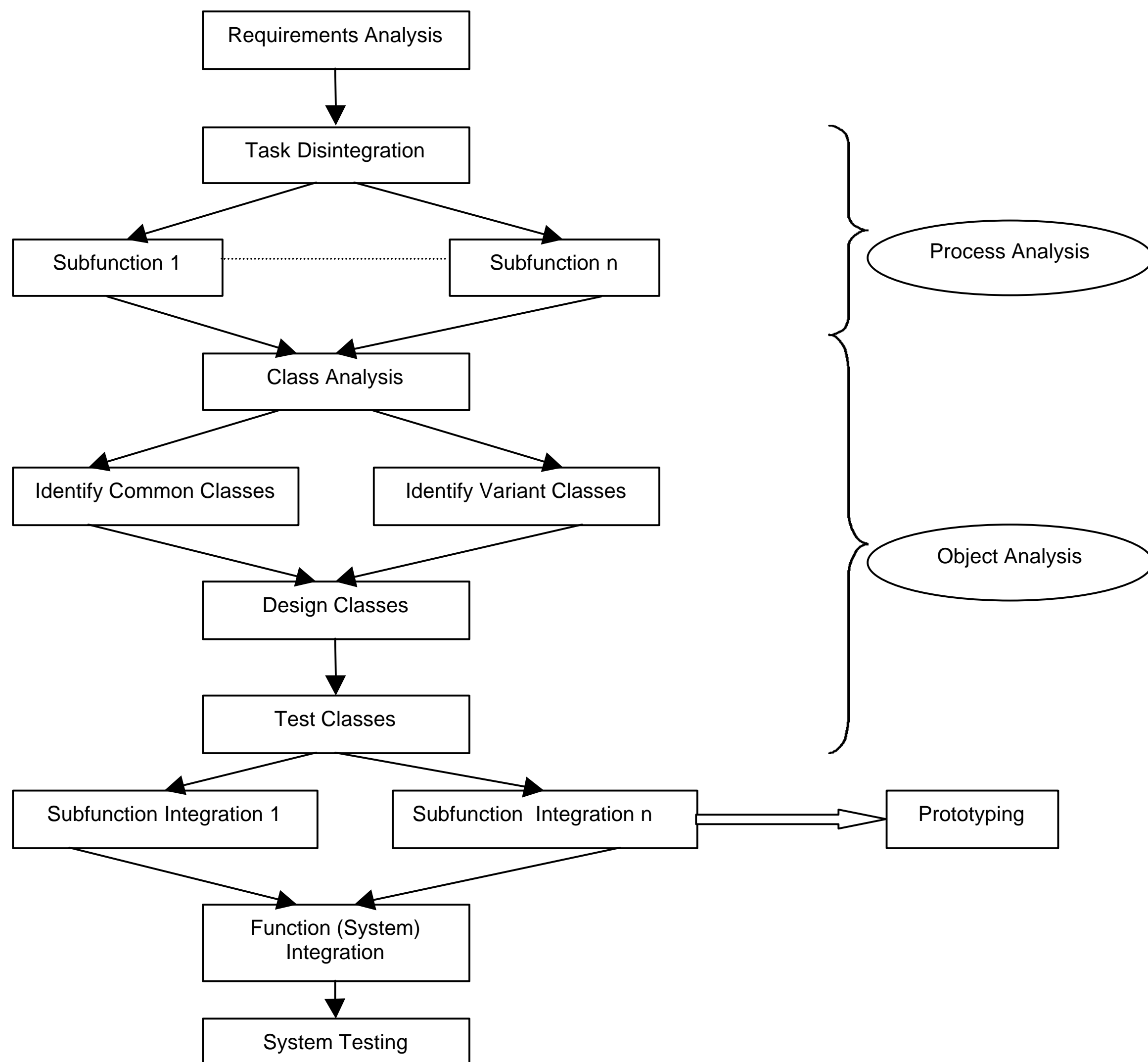

Fig. 1: Hybrid Convergence Model with Functional and Class Disintegration with Process and Analysis Phases

reuse them or reuse their specialized versions. This new methodology supports both evolution and synthesis of systems.

4.1 PROTOTYPING

The core idea that we have focused on when making the first design of the methodology presented in this paper is a continuous refinement through validation and verification not throughout the production but up to functional integration only. That is, by the time we reach the system integration stage there is no need for validation and verification with the users because the model asserts the requirement of complete and correct analysis and design before system integration. This is an absolute prerequisite. To reach this goal, several approaches were possible; formal methods, simulation or prototyping. We chose prototyping for the following reasons:

- Prototypes are partially developed product that enables customers and developers to examine some aspects of the proposed system and decide if it is suitable or appropriate for the finished product
- Implement a small portion of some key requirements to ensure that the requirements are consistent, feasible and practical; if not revisions are made at the requirement stage, rather than at the more costly testing stage
- Assess alternative design strategies and decide which is best for a particular project
- Prototyping is essential for producing correct software as well as promoting better communication between users and developers
- Prototyping is used for producing better user interfaces as well as for producing correct and understandable specifications
- It eliminates the discrepancies between the specifications and the implementations.

The hybrid convergence model allows to build very early and rapidly executable prototypes inspired from the OOAD method [9] – based on building real prototypes, each of them representing an agreed on step toward the final prototype. The prototype component in the figure, and not of rapid prototype (which is throwaway by definition) gives

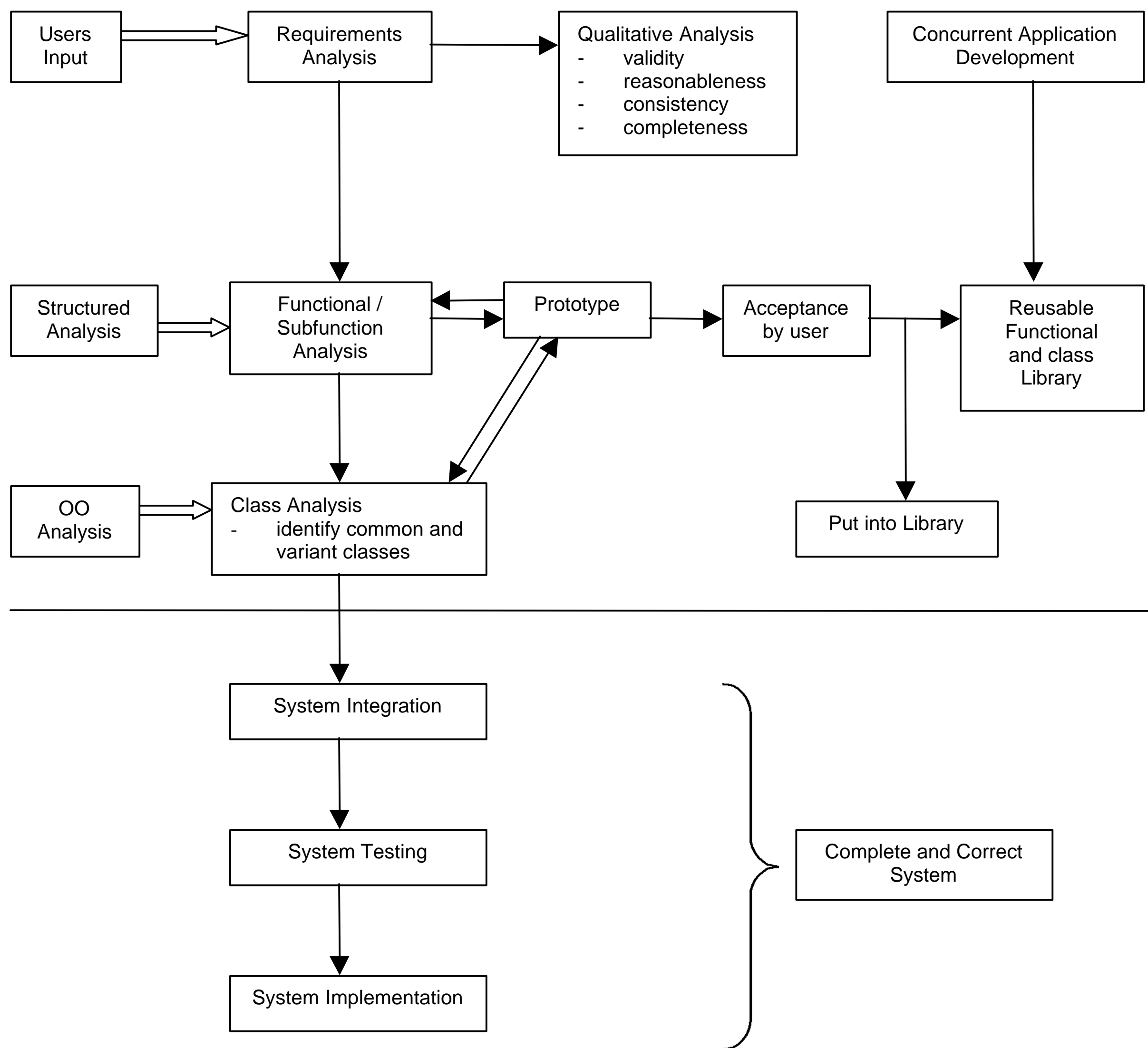

Fig. 2: Hybrid Convergence Life-Cycle Model

to these steps the main property of being an evolution i.e. a permanent ***convergence toward the final product. Therefore in this model, a prototype is being built and is converging toward the expected result while showing continuously that it is meeting the customer's requirements.*** All along the continuous development of the product through all these prototypes, we do not only plan for the refinement of code itself but also for the following (see also Fig. 3):

- A refinement of user requirements. In fact, each prototype will be defined according to a specific user requirement that we call prototype requirement. The converging evolution of the prototype requirement should lead to the sum of the originally specified user requirement and of its evolutions
- A refinement of specifications matching the prototype requirement.

The production of each functional and class prototype is composed of two generic and tightly coupled actions: "building/checking". These actions instantiation depends on the step we are at in the project cycle. Therefore, in this approach, a software life cycle is a kind of convergent stacking up of identical <building/checking> cycles. The result is a convergent "V" life cycle model in order to evoke convergence. A prototype is developed by following generalization life cycles. When these cycles stop, we obtain for the current step a stable state, which is the targeted prototype (See Fig. 4) [1].

4.2 USER INVOLVEMENT

There are many benefits from user involvement in application development. First, it builds commitment by users who automatically assume ownership of the system. Second, users who are the real experts at the jobs being automated are fully represented throughout development. Third, many tasks are performed by users, including design of screens, forms, and reports, development of user documentation, and development and conduct of

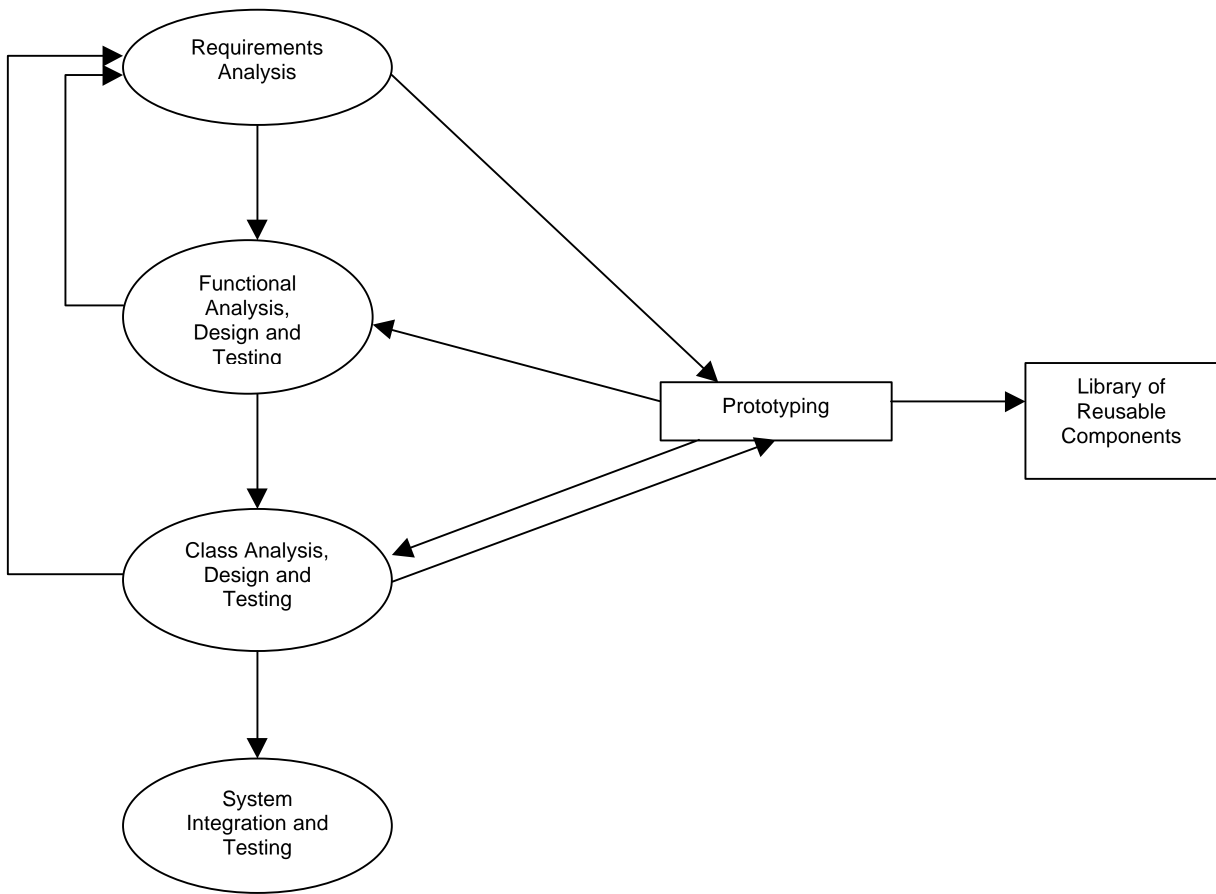

Fig. 3: General Phases of the Hybrid Convergence model

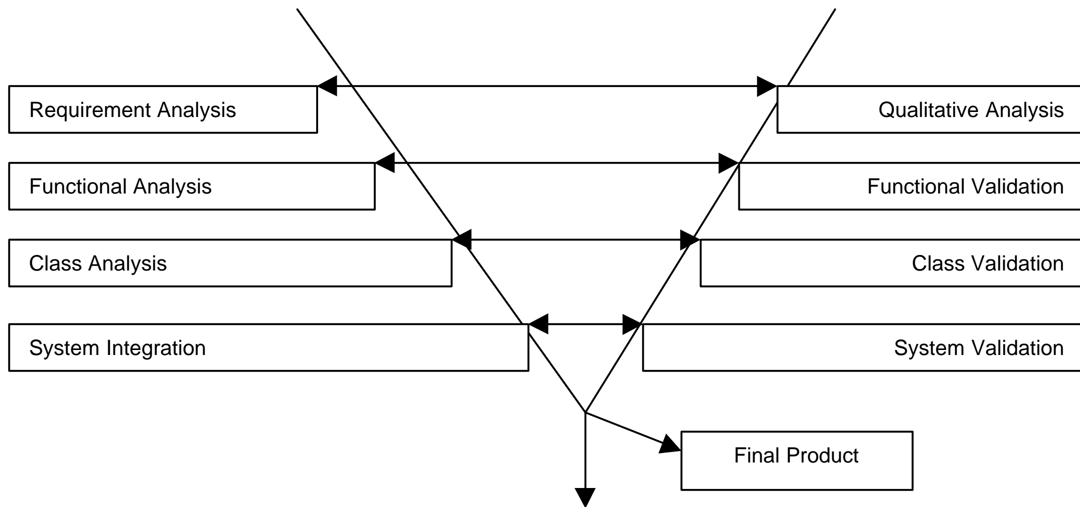

Fig. 4: Convergence to Final Product by Continuous Refinement

acceptance tests. There are two kinds of users in this model.

a) Users who are unfamiliar with technical and analysis jargons and,
b) Users who are familiar with systems analysis issues as well as analysis concepts.

We assume that user involvement is not only desirable, but also mandatory to truly effective application development product and process. The most important aspect of user involvement is that it must be meaningful. The users must be decision-makers and staff who fully understand the impact of their decisions, and who are interested in participating in the development process. User

involvement is absolutely necessary in the hybrid model. Software engineers and users who have participated in user-involved application development tend to be fully committed to user involvement as a requirement in application development [8].

5.0 JUSTIFICATIONS FOR THE HYBRID CONVERGENCE MODEL

The synthesis of two methodologies with the emphasis on amalgamation of their strengths should produce a model that has twice the strengths, that is the model should be better than the individual methods. The model filters the weaknesses that are found in both OO and structured methodologies and incorporates and amplifies the strengths of both methodologies. The model basically combines both process as well as object-oriented concepts. The "divide and conquer" approach used in this model closely resembles the concept of modularity and composability and the following strengths are derived from these concepts.

1. Two or more application systems can be developed at the same time by using available components in the library, thus supporting reusability.
2. Supports the reengineering of legacy systems from traditional structured methodology to OO methodology. Traditional systems would already have the functional decomposition aspect done before the system was implemented. Use the decomposition chart as a starting point for subfunctional and class analysis. Recode the classes using OO programming languages. It does not matter in what language the legacy systems are in. The model suggests that functions should be decomposed into subfunctions and then into classes.

3. It overcomes the problems of systems decomposition by focusing on functional decomposition at the outset of the analysis phase, thus improving system manageability particularly for large and complex systems.

4. Systems maintenance cost which occurs at later stage is greatly reduced due to the rigorous testing and prototyping methodology introduced at the early stages of systems development.

5. The model helps to control the thrashing by including activities and subprocesses that enhance under-standing by using prototyping. In thrashing developers may thrash from one activity to the next and then back again as they strive to gather knowledge about problems and how the proposed solution addresses it.

6. It forces developers to analyze systems in a domain specific context rather than in application specific context, thus encouraging at an early stage potentially reusable assets. Domain engineering reflects the idea that sharing between related applications occurs in one or more application domains – or problem domains or solution domains. It looks beyond a single system. Table 1 depicts at a glance a comparative summary of the three analysis methodologies.

6.0 CONCLUSION

Different systems can be developed in different ways using different methodologies. The hybrid paradigm proposed in the paper takes into consideration important factors such as speed of software production, re-engineering of legacy systems, reusability of components, concurrent system development and consistent user involvement in the development of systems. It is strongly encouraged that organizations that are engaged in hybrid development first experiment the model on a pilot medium sized project. With this new paradigm, it is hoped that many of the software engineering problems encountered by developers could be avoided.

REFERENCES

[1] Christopher ROUXEL, "Object-Oriented Methodologies for Large Scale Projects: does it work?" Journal of Object-Oriented Programming, 1994, pp. 84-86.

[2] Roger S. Pressman, "*Software Engineering – A Practitioner's Approach*", McGraw Hill 1997, p. 278.

[3] David Chun Sov, " Object-Oriented Case Tool Development Using The C++ Language", M. Buss. Sys, Monash University, March 1994, pp. 64-71.

[4] Schach, *Classical and Object-Oriented Software Engineering*, 3rd. Ed., IRWIN, 1996.

[5] I. Sommerville, *Software Engineering*, Addison-Wesley, 1992.

[6] L. C. Briand, V. R. Basili, Y. M. Kim, and D. R. Squier, "A Change Analysis Process to Characterise Software Maintenance Projects", *Proceedings of the International Conference on Software Maintenance*, Victoria, Canada, 1994.

[7] C. Ghezzi, M. Jazayeri and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, 1991.

[8] Sue Conger, *The New Software Engineering,*ITP 1994, pp. 39-40.

Table 1: Comparative Summary of the Three Methodologies

METHODOLOGY	***STRUCTURED***	***OBJECT-ORIENTED***	***HYBRID***
ANALYSIS	FUNCTIONAL AND DATA	CLASS AND DOMAIN	FUNCTIONAL, CLASS AND DOMAIN
REENGINEERING	DIRECT CONVERSION TO OO IS DIFFICULT	CONVERSION TO STRUCTURED IS NOT DIFFICULT	O-O-F TO F-O-O AND VICE-VERSA
ROBUSTNESS	DISRUPTION OF SERVICE DUE TO FUNCTIONAL COHESION	SAFE-STATE	SAFE-STATE
EXTENDIBILITY	DIFFICULT DUE TO APPLICATION SPECIFIC	EASIER DUE TO CLASS MODULARITY	EASIER DUE TO FUNCTIONAL AND CLASS MODULARITY
MAINTAINABILITY	DIFFICULT AND TIME-CONSUMING	EASIER DUE TO CLASS MODULARITY	LESS MAINTENANCE AT END-STAGE
REUSABILITY	FUNCTIONAL LIBRARIES	CLASS LIBRARIES	CLASS AND FUNCTIONAL LIBRARIES (CONCURRENT APPLICATION DEVELOPMENT)
INTEGRITY	ADDITIONAL SECURITY MEASURES REQUIRED	ENCAPSULATION AND INFORMATION HIDING (INHERENT)	ENCAPSULATION AND INFORMATION HIDING (INHERENT)
MODELING	HIGHER TO LOWER LEVEL (DECOMPOSITION)	CONSISTENT THROUGHOUT ANALYSIS AND DESIGN	HYBRID MODELING
SMOOTH TRANSFORMATION	CHECKPOINT PRODUCT	ANALYSIS TO DESIGN NOTATIONS ARE THE SAME	PHASEWISE REFINEMENT
CORRECTNESS	PROTOTYPING	PROTOTYPING AND CLASS REUSE	PROTOTYPING, CLASS REUSE AND USER VERIFICATION IS CONTINUOUS

[9] Booch, G. (1991). Object-Oriented Design: With Applications. Redwood City, CA, USA: Benjamin/Cummings.

[10] David M.Hilbert, Jason E.Robbins, David F.Redmiles, *EDEM: Intelligent Agents for Collecting Usage Data and Increasing User Involvement in Development,* Proceeding of the 1998 International Conference on Intelligent User Interfaces, Tasks and Usage, pp. 73-76, 1998.

BIOGRAPHY

Vasuthevan Balakrishnan obtained his B.S. in Computer Science in 1985 from the University of Southern California and Masters in Software Engineering from the University of Malaya in 1999. He is currently lecturing at Monash University Sunway Campus in the area of Computer Science.

Sai Peck Lee obtained her Master of Computer Science from University of Malaya in 1990, her D.E.A of Computer Science from University of Paris VI Pierre et Marie Curie in 1991 and her Ph.D of Computer Science from University of Paris I Pantheon-Sorbonne in 1994. She is a lecturer at Faculty of Computer Science and Information Technology, University of Malaya.