CHAPTER 4- IT THEORY: TOOLS, METHODS, AND APPLICATIONS – II

4.1 Methods of Transformation (contd.)

In this chapter, I continue discussing the methods of transformation. I examine, in particular, (1) the legacy application services including migration, (2) product life cycle management (PLM), (3) quality services, (4) system integration services, (5) enterprise application integration, (6) infrastructure related services, (7) testing services, (8) server oriented architecture (SOA), and (9) compliance.

4.1.1 LEGACY SYSTEMS AND DATA MIGRATION

Legacy systems are considered to be potentially problematic by many software engineers *(Bisbal, 1999)*. Legacy systems often run on obsolete (and usually slow) hardware, and sometimes spare parts for such computers become increasingly difficult to obtain. These systems are often hard to maintain, improve, and expand because there is a general lack of understanding of the system. The designers of the system may have left the organization, leaving no one left to explain how it works. Such a lack of understanding can be exacerbated by inadequate documentation or manuals getting lost over the years. Integration with newer systems may also be difficult because new software may use completely different technologies.

Despite these problems, organizations can have compelling reasons for keeping a legacy system, such as:

- The costs of redesigning the system are prohibitive because it is large, monolithic, and/or complex.
- The system requires close to 100% availability, so it cannot be taken out of service, and the cost of designing a new system with a similar availability level is high.

- The way the system works is not well understood. Such a situation can occur when the designers of the system have left the organization and the system has either not been fully documented or such documentation has been lost.

- The user expects that the system can easily be replaced when this becomes necessary.

- The system works satisfactorily, and the owner sees no reason for changing it; or in other words, re-learning a new system would have a prohibitive attendant cost in lost time and money.

If legacy software runs on only antiquated hardware, the cost of maintaining the system may eventually outweigh the cost of replacing both the software and hardware unless some form of emulation or backward compatibility allows the software to run on new hardware. However, many of these systems do still meet the basic needs of the organization. The systems to handle customers' accounts in banks are one example. Therefore the organization cannot afford to stop them and yet some cannot afford to update them. A demand of extremely high availability is commonly the case in computer reservation systems, air traffic control, energy distribution (power grids), nuclear power plants, military defense installations, and other systems critical to safety, security, traffic throughput, and/or economic profits. The change being undertaken in some organizations is to switch to Automated Business Process (ABP) software which generates complete systems. These systems can then interface to the organizations' legacy systems and use them as data repositories. This approach can provide a number of significant benefits: the users are insulated from the inefficiencies of their legacy systems, and the changes can be incorporated quickly and easily in the ABP software.

Note that "legacy" has little to do with the size or even age of the system — mainframes run 64-bit Linux and Java, after all, right alongside 1960s vintage code. In fact, some of the thorniest legacy problems organizations now face are in trying to leverage or replace existing "fat client" Visual Basic code as customers demand reliable Web accessible systems. The term legacy support is also often used with reference to obsolete or "legacy" computer hardware,

whether peripherals or core components. Operating systems with "legacy support" can detect and use legacy hardware.

The term is also used as a verb for what vendors do for products in legacy mode - they "support", or provide software maintenance, for obsolete or "legacy" products. In some cases, "legacy mode" refers more specifically to backward compatibility. The computer mainframe era saw many applications running in legacy mode. In the modern business computing environment, n-tier, or 3-tier architectures are more difficult to place into legacy mode as they include many components making up a single system. Government regulatory changes must also be considered in a system running in legacy mode. Virtualization technology allows for a resurgence of modern software applications entering legacy mode. As system complexity and software costs increase, many computing users keep their current systems permanently in legacy mode.

There is an alternate point of view - growing since the "Dot Com" bubble burst in 1999 - that legacy systems are simply (and only) computer systems that are both installed and working. In other words, the term is not at all pejorative - quite the opposite. Perhaps the term "legacy" is only an effort by computer industry salesmen to generate artificial churn in order to encourage purchase of unneeded technology. IT analysts estimate that the cost to replace business logic is about five times that of reuse, and that's not counting the risks involved in wholesale replacement. Shareholders and managers are increasingly asking the reasons for so much money being spent on new technology with so little to show for it. Ideally businesses would never have to rewrite most core business logic. After all, debits must equal credits - they always have, and they always will. Businesses and governments are also flinching at well-publicized system failures and security breaches that all too commonly arrive with new software - failures which are utterly catastrophic in many cases. There's also a growing backlash against large, packaged software products (SAP, Oracle, PeopleSoft, and others) which were oversold and in some cases have proven too costly, inflexible, and poorly matched to business needs.

Increasingly the IT industry is responding to these understandable business concerns. "Legacy modernization" and "legacy transformation" are now popular terms, and they mean reusing and refactoring existing, core business logic by providing new user interfaces (typically Web

interfaces) and service-enabled access (e.g., through Web services). These techniques allow organizations to understand their existing code assets (using discovery tools), provide new user and application interfaces to existing code, improve workflow, contain costs, minimize risk, and enjoy classic qualities of service (near 100% uptime, security, scalability, etc.). Technology companies involved in "enterprise transformation" are growing and profiting by what many people feel is a more rational approach toward legacy systems.

The re-examination of attitudes toward legacy systems is also inviting more reflection on what makes legacy systems as durable as they are. Technologists are relearning the fact that sound architecture, practiced up front, helps businesses avoid costly and risky rewrites in the first place. The most common legacy systems tend to be those which embraced well-known IT architectural principles, with careful planning and strict methodology during implementation. Poorly designed systems often don't last. Thus, many organizations are rediscovering not only the value in the legacy systems themselves but also their philosophical underpinnings.

Data migration, on the other hand is the process of transferring data between storage types, formats, or computer systems. Data migration is usually performed programmatically to achieve an automated migration, freeing up human resources from tedious tasks. It is required when organizations or individuals change computer systems or upgrade to new systems. To achieve an effective data migration procedure, data on the old system is mapped to the new system providing a design for data extraction and data loading. The design relates old data formats to the new system's formats and requirements. Programmatic data migration may involve many phases but it minimally includes data extraction where data is read from the old system and data loading where data is written to the new system. After loading into the new system, results are subjected to data verification to determine that data was accurately translated, is complete, and supports processes in the new system. During verification, there may be a need for a parallel run of both systems to identify areas of disparity and forestall erroneous data loss. Automated and manual data cleansing is commonly performed in migration to improve data quality, eliminate redundant or obsolete information, and match the requirements of the new system. Data migration phases (design, extraction, cleansing, load, verification) for applications of moderate to high complexity are commonly repeated several times before the new system is activated.

4.1.1.1 Product Lifecycle Management

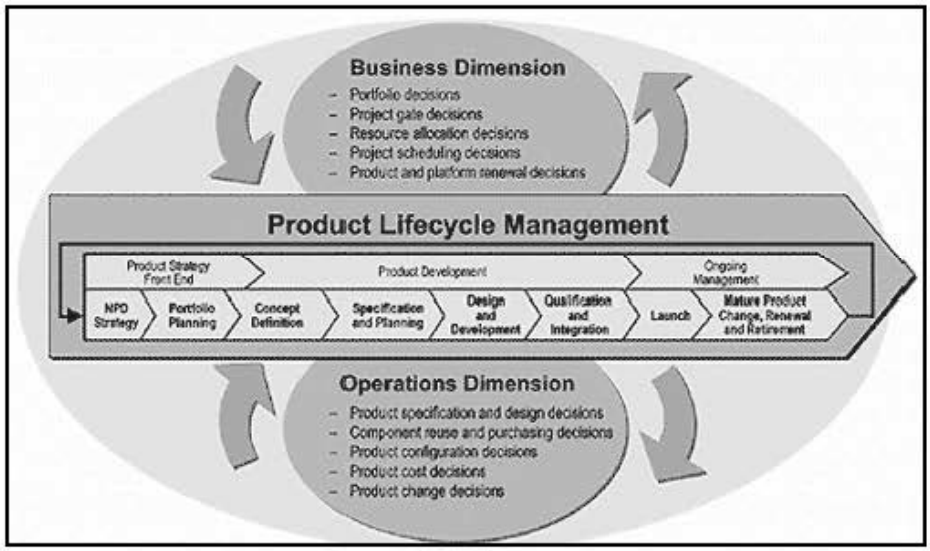
Product lifecycle management (PLM) is the process of managing the entire lifecycle of a product from its conception, through design and manufacture, to service and disposal *(CiMData, 2002)*. It is one of the four cornerstones of a corporation's information technology structure *(Evans, 2004)*. All companies need to manage communications and information with their customers (CRM-Customer Relationship Management) and their suppliers (SCM-Supply Chain Management) and the resources within the enterprise (ERP-Enterprise Resource Planning). In addition, manufacturing engineering companies must also develop, describe, manage and communicate information about their products (PLM).

Documented benefits include *(Hill, 2006)*:

1. Reduced time to market
2. Improved product quality
3. Reduced prototyping costs
4. Savings through the re-use of original data
5. A framework for product optimization
6. Reduced waste
7. Savings through the complete integration of engineering workflows

Product Lifecycle Management (PLM) as depicted in diagram 4.1 is more to do with managing descriptions and properties of a product through its development and useful life, mainly from a business/engineering point of view; whereas Product life cycle management (PLCM) is to do with the life of a product in the market with respect to business/commercial costs and sales measures *(Ausura, 2007)*.

Diagram 4.1 - Product Lifecycle Management



Product lifecycle management (PLM) as depicted in diagram 4.1 is the title commonly applied to a set of application software that enables the New Product Development (NPD) business process.

Within PLM there are four primary areas;

1. Product and Portfolio Management (PPM)
2. Product Design (CAx)
3. Manufacturing Planning (MPM)
4. Product Data Management (PDM)

Note: While application software is not required for PLM processes, the business complexity and rate of change requires organizations execute as rapidly as possible.

Product Data Management is focused on capturing and maintaining information on products and/or services through its development and useful life. Product and Portfolio Management is focused on managing resource allocation, tracking the progress vs. plan for projects in the new

product development projects that are in process (or in a holding status). Portfolio management is a tool that assists management in tracking progress on new products and making trade-off decisions when allocating scarce resources. The core of PLM (product lifecycle management) is in the creation and central management of all product data and the technology used to access this information and knowledge. PLM as a discipline emerged from tools such as CAD, CAM and PDM, but can be viewed as the integration of these tools with methods, people and the processes through all stages of a product's life *(Teresco, 2004)*. It is not just about software technology but is also a business strategy *(Stackpole, 2003)*.

For simplicity the stages described are shown in a traditional sequential engineering workflow. The exact order of event and tasks will vary according to the product and industry in question but the main processes are *(Gould, 2002)*.

- Conceive
  - ✓ Specification
  - ✓ Concept design
- Design
  - ✓ Detailed design
  - ✓ Validation and analysis (simulation)
  - ✓ Tool design
- Realize
  - ✓ Plan manufacturing
  - ✓ Manufacture
  - ✓ Build/Assemble
  - ✓ Test (quality check)
- Service
  - ✓ Sell and Deliver
  - ✓ Use
  - ✓ Maintain and Support
  - ✓ Dispose

The major key point events are:

a) Order
b) Idea
c) Kick-off
d) Design freeze
e) Launch

The reality is however more complex, people and departments cannot perform these tasks in isolation and one activity cannot simply finish and the next activity start. Design is an iterative process, often designs need to be modified due to manufacturing constraints or conflicting requirements. Where exactly a customer order fits into the time line depends on the industry type, whether the products are for example Build to Order, Engineer to Order, or Assemble to Order.

The Product Lifecycle has the following phases - Many software solutions have developed to organize and integrate the different phases of a product's lifecycle. PLM should not be seen as a single software product but a collection of software tools and working methods integrated together to address either single stages of the lifecycle or connect different tasks or manage the whole process. Some software providers cover the whole PLM range while others a single niche application. Some applications can span many fields of PLM with different modules within the same data model. Here, we provide an overview of the fields within PLM. It should be noted however that the simple classifications do not always fit exactly, many areas overlap and many software products cover more than one area or do not fit easily into one category. It should also not be forgotten that one of the main goals of PLM is to collect knowledge that can be reused for other projects and to coordinate simultaneous concurrent development of many products. PLM is about business processes, people and methods as much as software application solutions. Although PLM is mainly associated with engineering tasks it also involves marketing activities such as Product Portfolio Management (PPM), particularly with regards to New Product Introduction (NPI).

Phase 1: Conccivc – Imaginc, Spccify, Plan, Innovatc

The first stage in idea is the definition of its requirements based on customer, company, market and regulatory bodies' viewpoints. From this a specification of the products major technical parameters can be defined. Although often this task is carried out using standard office software packages there are for the field of requirements management a number of specialized software tools available.

Parallel to the requirements specification the initial concept design work is carried out defining the visual aesthetics of the product together with its main functional aspects. For the Industrial Design, Styling, work many different medias are used from pencil and paper, clay models to 3D CAID Computer-aided industrial design software.

Phase 2: Design – Describe, Define, Develop, Test, Analyze and Validate

Phase 2 is where the detailed design and development of the product's form starts, progressing to prototype testing, through pilot release to full product launch. It can also involve redesign and ramp for improvement to existing products as well as planned obsolescence. The main tool used for design and development is CAD Computer-aided design. This can be simple 2D Drawing / Drafting or 3D Parametric Feature Based Solid/Surface Modeling, Such software includes technology such as Hybrid Modeling, Reverse Engineering, KBE (Knowledge-Based Engineering), NDT (Non-destructive testing), and Assembly construction.

This step covers many engineering disciplines including: Mechanical, Electrical, Electronic, Software (embedded), and domain-specific, such as Architectural, Aerospace, Automotive ... Along with the actual creation of geometry there is the analysis of the components and product assemblies. Simulation, validation and optimization tasks are carried out using CAE (Computer-aided engineering) software either integrated in the CAD package or stand-alone. These are used to perform tasks such as:- Stress analysis, FEA (Finite Element Analysis); Kinematics; Computational fluid dynamics (CFD); and mechanical event simulation (MES). CAQ (Computer-aided quality) is used for tasks such as Dimensional Tolerance (engineering) Analysis. Another task performed at this stage is the sourcing of bought out components, possibly with the aid of Procurement systems.

Phase 3: Realize – Manufacture, Make, Build, Procure, Produce, Sell and Deliver

Once the design of the product's components is complete the method of manufacturing is defined. This includes CAD tasks such as tool design; creation of CNC Machining instructions for the product's parts as well as tools to manufacture those parts, using integrated or separate CAM Computer-aided manufacturing software. This will also involve analysis tools for process simulation for operations such as casting, molding, and die press forming. Once the manufacturing method has been identified MPM – (Manufacturing Process Management) comes into play. This involves CAPE (Computer-aided Production Engineering) or CAP/CAPP – (Production Planning) tools for carrying out Factory, Plant and Facility Layout and Production Simulation. For example: Press-Line Simulation; and Industrial Ergonomics; as well as tool selection management. Once components are manufactured, their geometrical form and size can be checked against the original CAD data with the use of Computer Aided Inspection equipment and software. Parallel to the engineering tasks, sales product configuration and marketing documentation work will be taking place. This could include transferring engineering data (geometry and part list data) to a web based sales configurator and other Desktop Publishing systems.

Phase 4: Service – Use, Operate, Maintain, Support, Sustain, Phase-out, Retire, Recycle and Disposal

The final phase of the lifecycle involves managing of in service information. Providing customers and service engineers with support information for repair and maintenance, as well as waste management/recycling information. This involves using such tools as Maintenance, Repair and Operations Management (MRO) software.

All phases: product lifecycle – Communicate, Manage and Collaborate

None of the above phases can be seen in isolation. In reality a project does not run sequentially or in isolation of other product development projects. Information is flowing between different people and systems. A major part of PLM is the co-ordination of and management of product definition data. This includes managing engineering changes and

release status of components; configuration product variations; document management; planning project resources and timescale and risk assessment.

For these tasks graphical, text and metadata such as product BOMs (Bill of Materials) needs to be managed. At the engineering departments level this is the domain of PDM – (Product Data Management) software, at the corporate level EDM (Enterprise Data Management) software, these two definitions tend to blur; however, it is typical to see two or more data management systems within an organization. These systems are also linked to the other corporate systems such as SCM, CRM, and ERP. Associated with these systems are Project Management Systems for Project/Program Planning.

This central role is covered by numerous Collaborative Product Development tools which run throughout the whole lifecycle and across organizations. It requires many technology tools in the areas of Conferencing, Data Sharing and Data Translation. The field being Product visualization which includes technologies such as DMU (Digital Mock-Up), Immersive Virtual Digital prototyping (Virtual reality) and Photo realistic Imaging.

Product Development processes and methodologies

A number of established methodologies have been adopted by PLM and been further advanced. Together with PLM digital engineering techniques, they have been advanced to meet company goals such as reduced time to market and lower production costs. Reducing lead times is a major factor as getting a product to market quicker than the competition will help with higher revenue and profit margins and increase market share.

These techniques include:-

- Concurrent engineering workflow
- Industrial Design
- Bottom-up design
- Top-down design

- Front loading design workflow
- Design in context
- Modular design
- NPD New product development
- Digital simulation engineering
- Requirement driven design
- Specification managed validation

Concurrent engineering workflow – This is a workflow that instead of working sequentially through stages carries out a number of tasks in parallel. For example: starting tool design before the detailed designs of the product are finished, or the engineer starting on detail design solid models before the concept design surfaces models are complete. Although this does not necessarily reduce the amount of manpower required for a project, it does drastically reduce lead times and thus time to market. Feature based CAD systems have for many years allowed the simultaneous work on 3D solid model and the 2D drawing by means of 2 separate files, with the drawing looking at the data in the model; when the model changes the drawing will associatively update. Some CAD packages also allow associative copying of geometry between files. This allows, for example, the copying of a part design into the files used by the tooling designer. The manufacturing engineer can then start work on tools before the final design freeze; when a design changes size or shape the tool geometry will then update. Concurrent engineering also has the added benefit of providing better and more immediate communication between departments, reducing the chance of costly, late design changes. It adopts a problem prevention method as compared to the problem solving and re-designing method of traditional sequential engineering.

Bottom-up design – Bottom-up design (CAD Centric) is where the definition of 3D models of a product starts with the construction of individual components. These are then virtually

brought together in sub-assemblies of more than one level until the full product is digitally defined. This is sometimes known as the review structure showing what the product will look like. The BOM contains all of the physical (solid) components; it may (but not also) contain other items required for the final product BOM such as paint, glue, oil and other materials commonly described as 'bulk items'. Bulk items typically have mass and quantities but are not usually modeled with geometry.

Top-down design – Top-down design (Part Centric) follows closer the true design process. This starts with a layout model, often a simple 2D sketch defining basic sizes and some major defining parameters. Industrial Design brings creative ideas to product development. Geometry from this is associatively copied down to the next level, which represents different sub-systems of the product. The geometry in the sub-systems is then used to define more detail in levels below. Depending on the complexity of the product, a number of levels of this assembly are created until the basic definition of components can be identified, such as position and principal dimensions. This information is then associatively copied to component files. In these files the components are detailed; this is where the classic bottom-up assembly starts. The top down assembly is sometime known as a control structure. If a single file is used to define the layout and parameters for the review structure it is often known as a skeleton file.

Defense engineering traditionally develops the product structure from the top down. The system engineering process prescribes a functional decomposition of requirements and then physical allocation of product structure to the functions. This top down approach would normally have lower levels of the product structure developed from CAD data as a bottom up structure or design.

Front loading design and workflow – Front loading is taking top-down design to the next stage. The complete control structure and review structure, as well as downstream data such as drawings, tooling development and CAM models, are constructed before the product has been defined or a project kick-off has been authorized. These assemblies of files constitute a template from which a family of products can be constructed. When the decision has been made to go with a new product, the parameters of the product are entered into the template model and all the associated data is updated. Obviously, predefined associative models will not

be able to predict all possibilities and will require additional work. The assumption is that a lot of the experimental/investigative work has already been completed. A lot of knowledge is built into these templates to be reused on new products. This does require additional resources "up front" but can drastically reduce the time between project kick-off and launch. Such methods do however require organizational changes, as considerable engineering efforts are moved into "offline" development departments. It can be seen as an analogy to creating a concept car to test new technology for future products, but in this case the work is directly used for the next product generation.

Design in context – Individual components cannot be constructed in isolation. CAD models of components are designed within the context of part or the entire product being developed. This is achieved using assembly modeling techniques. Other components' geometry can be seen and referenced within the CAD tool being used. The other components within the sub-assembly, may or may not have been constructed in the same system, their geometry being translated from other CPD formats. Some assembly checking such as DMU is also carried out using Product visualization software.

Major commercial players – Total spending on PLM software and services is estimated to be above $15 billion a year but it is difficult to find any two market analysis reports that agree on figures *(CIMdata, Oct 2006), (Daratech, Mar 2006).* Market growth estimates are in the 10% area. Looking at segment split, currently most of the revenue generated is in the area of EDA and high end MCAD (each above 15%), followed by AEC, low-end MCAD, and PDM (each above 10%). The other notable segment is CAE at above 5%. It is however predicted that the collaborative PDM and visualization areas will increase in dominance.

There are many companies that supply software to support the PLM process; the largest by revenue are mentioned here. Some companies such as Siemens PLM Software ($1.1B), Altair Engineering Inc. ($0.15B), Dassault Systèmes ($1.1B), Agile Software Corporation (recently acquired by Oracle Corporation) and SofTech, Inc. (.011B) provide software products that cover most of the areas of PLM functionality; some like PTC ($0.8B) cover a number of segments; other companies for example MSC Software ($0.3B) provide packages specializing in specific topics. One company, Aras Corp offers Microsoft-based open source enterprise

PLM solutions *(Stackpole, 2007)* and another Arena Solutions, provides on-demand PLM software. Additional unique offerings include Selerant which specializes only in the process industry and provides formulation optimization and regulatory management. Also, Omnify Software's PLM incorporates traditionally disparate systems (quality, training, corrective action/preventive action) to augment support for regulatory compliance across all verticals , provides a web-based PLM solution called XpressCommerce that mainly services apparel and footwear retailers and brand manufacturers *(Swain, Mar 2007).* Independent PLM service providers such as SIA Conseil, Accenture, Integware and Metafore deliver PLM consulting services by providing information to help companies plan and implement PLM practices, processes and technologies.

There are also companies whose main revenue is not from PLM but do attribute some of their income from PLM software, such as SAP ($11B), SSA Global , Oracle Corporation and Autodesk ($1.5B). Other companies in this market, such as IBM ($88.9B), EDS ($19.8B), Accenture, Infosys (INFY), geometricglobal.com, Tata Consultancy Services (TCS) and, ITC Infotech provide outsourcing and consulting services some of which is in the field of PLM.

4.1.2 QUALITY SERVICES

In the context of software engineering, software quality measures how well software is designed (quality of design), and how well the software conforms to that design (quality of conformance) *(Pressman, 2005)*, although there are several different definitions. Whereas quality of conformance is concerned with implementation (see Software Quality Assurance), quality of design measures how valid the design and requirements are in creating a worthwhile product *(Pressman, 2005).*

A definition in Steve McConnell's Code Complete similarly divides software into two pieces: internal and external quality characteristics. External quality characteristics are those parts of a product that face its users, where internal quality characteristics are those that do not *(McConnel, 1993).* Another definition by Dr. Tom DeMarco says "a product's quality is a function of how much it changes the world for the better" *(DeMarco, 1999).* This can be interpreted as meaning that user satisfaction is more important than anything in determining software quality. Another

definition, coined by Gerald Weinberg in Quality Software Management: Systems Thinking is "Quality is value to some person." This definition stresses that quality is inherently subjective - different people will experience the quality of the same software very differently. One strength of this definition is the questions it invites software teams to consider, such as "Who are the people we want to value our software?", and "What will be valuable to them?"

Software reliability is an important facet of software quality. It is defined as "the probability of failure-free operation of a computer program in a specified environment for a specified time" *(Musa, 1987)*. One of reliability's distinguishing characteristics is that it is objective, measurable, and can be estimated, whereas much of software quality has subjective criteria. This distinction is especially important in the discipline of Software Quality Assurance. These measured criteria are typically called software metrics.

A software quality factor is a non-functional requirement for a software program which is not called up by the customer's contract, but nevertheless is a desirable requirement which enhances the quality of the software program.

4.1.2.1 Quality Factors

There can be several quality factors. Some of these are elaborated below.

Understandability is the characteristic which a software product possesses if the purpose of the product is clear. This goes further than just a statement of purpose - all of the design and user documentation must be clearly written so that it is easily understandable. This is obviously subjective in that the user context must be taken into account, i.e. if the software product is to be used by software engineers it is not required to be understandable to the layman.

A software product possesses the characteristic completeness to the extent that all of its parts are present and each of its parts is fully developed. This means that if the code calls a sub-routine from an external library, the software package must provide reference to that library and all required parameters must be passed. All required input data must be available.

A software product possesses the characteristic conciseness to the extent that no excessive information is present. This is important where memory capacity is limited, and it is

important to reduce lines of code to a minimum. It can be improved by replacing repeated functionality by one sub-routine or function which achieves that functionality. It also applies to documents.

A software product possesses the characteristic portability to the extent that it can be operated easily and well on multiple computer configurations. Portability can mean both between different hardware setups--such as running on a Mac as well as a PC--and between different operating systems--such as running on both Mac OS X and GNU/Linux.

A software product possesses the characteristic consistency to the extent that it contains uniform notation, symbology and terminology within itself.

A software product possesses the characteristic maintainability to the extent that it facilitates updating to satisfy new requirements. Thus the software product which is maintainable should be well-documented, not complex, and should have spare capacity for memory usage and processor speed.

A software product possesses the characteristic testability to the extent that it facilitates the establishment of acceptance criteria and supports evaluation of its performance. Such a characteristic must be built-in during the design phase if the product is to be easily testable - a complex design leads to poor testability.

A software product possesses the characteristic usability to the extent that it is convenient and practicable to use. This is affected by such things as the human-computer interface. The component of the software which has most impact on this is the user interface (UI), which for best usability is usually graphical (i.e. a GUI).

A software product possesses the characteristic reliability to the extent that it can be expected to perform its intended functions satisfactorily. This implies a time factor in that a reliable product is expected to perform correctly over a period of time. It also encompasses environmental considerations in that the product is required to perform correctly in whichever conditions it finds itself - this is sometimes termed robustness.

A software product possesses structuredness to the extent that it possesses a definite pattern of organisation in its constituent parts. A software product written in a block-structured language such as Pascal will satisfy this characteristic.

A software product possesses the characteristic efficiency to the extent that it fulfills its purpose without waste of resources. This means resources in the sense of memory utilization and processor speed.

A software product possesses the characteristic security to the extent that it is able to protect data against unauthorized access and to withstand malicious interference with its operations. Besides presence of appropriate security mechanisms such as authentication, access control and encryption, security also implies reliability in the face of malicious, intelligent and adaptive attackers.

4.1.2.2 Measurement of software quality factors

There are varied perspectives within the field on measurement. There are a great many measures that are valued by some professionals, or in some contexts, that are decried as harmful by others. Some believe that quantitative measures of software quality are essential. Others believe that contexts where quantitative measures are useful are quite rare, and so prefer qualitative measures. Several leaders in the field of software testing have written about the difficulty of measuring what we truly want to measure well *(Hoffman, 2000).*

One example of a popular metric is the number of faults encountered in the software. Software that contains few faults is considered by some to have higher quality than software that contains many faults. Questions that can help determine the usefulness of this metric in a particular context include:

What constitutes 'many faults'? Does this differ depending on the purpose of the software (e.g. blogging software v. navigational software)? Does this take into account the size and complexity of the software? Does this account for the importance of the bugs (and the importance to the stakeholders of the people those bugs bug)? Does one try to weight this measure by the severity of the fault, or the incidence of the users it affects? If so, how? And if not, how does one know that 100 faults discovered is better than 1000? If the count of faults

being discovered is shrinking, how does one know what that means? For example, does that mean that the product is now higher quality than it was before? Or that this is a smaller/less ambitious change than before? Or that less tester-hours have gone into the project than before? Or that this project was tested by less skilled testers than before? Or that the team has discovered that less faults reported is in their interest?

This last question points to an especially difficult one to manage. All software quality metrics are in some sense measures of human behavior, since humans create software. If a team discovers that they will benefit from a drop in the number of reported bugs, there is a strong tendency for the team to start reporting less defects. That may mean that email begins to circumvent the bug tracking system, or that four or five bugs get lumped into one bug report, or that testers learn not to report minor annoyances. The difficulty is measuring what we mean to measure, without creating incentives for software programmers and testers to consciously or unconsciously "game" the measurements.

Software Quality Factors cannot be measured because of their vague description. It is necessary to find measures, or metrics, which can be used to quantify them as non-functional requirements. For example, reliability is a software quality factor, but cannot be evaluated in its own right. However there are related attributes to reliability, which can indeed be measured. Such attributes are mean time to failure, rate of failure occurrence, availability of the system. Similarly, an attribute of portability is the number of target dependent statements in a program.

A scheme which could be used for evaluating software quality factors is given below. For every characteristic, there are a set of questions which are relevant to that characteristic. Some type of scoring formula could be developed based on the answers to these questions, from which a measure of the characteristic may be obtained.

Understandability – Are variable names descriptive of the physical or functional property represented? Do uniquely recognizable functions contain adequate comments so that their purpose is clear? Are deviations from forward logical flow adequately commented? Are all elements of an array functionally related?

Completeness – Are all of its parts present and each of them fully developed?

Conciscncss – Are all the codes reachable? Is any code redundant? How many statements within loops could be placed outside the loop, thus reducing computation time? Are branch decisions too complex?

Portability – Does the program depend upon system or library routines unique to a particular installation? Have machine-dependent statements been flagged and commented? Has dependency on internal bit representation of alpha-numeric or special characters been avoided? How much effort is required to transfer the program from one hardware/software system environment to another?

Consistency – Is one variable name used to represent different physical entities in the program? Does the program contain only one representation for physical or mathematical constants? Are functionally similar arithmetic expressions similarly constructed? Is a consistent scheme for indentation used?

Maintainability – Has some memory capacity been reserved for future expansion? Is the design cohesive, i.e., each module has recognizable functionality? Does the software allow for a change in data structures (object-oriented designs are more likely to allow for this)? If a functionally-based design (rather than object-oriented), is a change likely to require restructuring the main-program, or just a module?

Testability – Are complex structures employed in the code? Does the detailed design contain clear pseudo-code? Is the pseudo-code at a higher level of abstraction than the code? If tasking is used in concurrent designs, are schemes available for providing adequate test cases?

Usability – Is a GUI used? Is there adequate on-line help? Is a user manual provided? Are meaningful error messages provided?

Reliability –Are loop indexes range tested? Is input data checked for range errors? Is divide-by-zero avoided? Is exception handling provided? What is the extent to which a program can be expected to perform its intended function with rescission?

Structuredness – Is a block-structured programming language used? Are modules limited in size? Have the rules for transfer of control between modules been established and followed?

Efficiency –Have functions been optimized for speed? Have repeatedly used blocks of code been formed into sub-routines? Checked for any memory leak, overflow? How much amount of computing resources and codes is required by a program to perform its function?

Security – Does the software protect itself and its data against unauthorized access and use? Does it allow its operator to enforce security policies? Are appropriate security mechanisms in place? Are those security mechanisms implemented correctly? Can the software withstand attacks that must be expected in its intended environment? Is the software free of errors that would make it possible to circumvent its security mechanisms? Does the architecture limit the impact of yet unknown errors? Security testing in any developed system is about finding loops and weaknesses of the system.

User's perspective – In addition to the technical qualities of software, the end user's experience also determines the quality of software. This aspect of software quality is called usability. It is hard to quantify the usability of a given software product. Some important questions to be asked are:

- Is the user interface intuitive?
- Is it easy to perform easy operations?
- Is it feasible to perform difficult operations?
- Does the software give sensible error messages?
- Do widgets behave as expected?
- Is the software well documented?
- Is the user interface self-explanatory/ self-documenting?
- Is the user interface responsive or too slow?

Last but not the least, the availability of (free or paid) support may determine the usability of the software. This is especially important in today's context – where the open source software (OSS) is gaining momentum with the advent of Linux. Entire companies have been formed that specialize in providing support services to the otherwise free OS. IBM Global services, for example, has given Linux its full backing. Likewise, the other consulting firms in this space also have their specialization areas as well.

4.1.3 SYSTEMS INTEGRATION

System integration is the bringing together of the component subsystems into one system and ensuring that the subsystems function together as a system. In information technology, systems integration is the process of linking together different computing systems and software applications physically or functionally. The system integrator brings together discrete systems utilizing a variety of techniques such as computer networking, enterprise application integration, business process management or manual programming.

A system is an aggregation of subsystems cooperating so that the system is able to deliver the overall functionality. System integration involves integrating existing (often disparate) subsystems. The subsystems will have interfaces. Integration involves joining the subsystems together by "gluing" their interfaces together. If the interfaces don't directly interlock, the "glue" between them can provide the required mappings. System integration is about determining the required "glue". System integration is also about value-adding to the system, capabilities that are possible because of interactions between subsystems.

In today's connected world, the role of system integration engineers is becoming more and more important: more and more systems are designed to connect together, both within the system under construction and to systems that are already deployed.

A system integration engineer needs a broad range of skills and is likely to be defined by a breadth of knowledge rather than a depth of knowledge. These skills are likely to include software and hardware engineering, interface protocols, and general problem solving skills. It is likely that the problems to be solved have not been solved before except in the broadest sense.

They are likely to include new and challenging problems with an input from a broad range of engineers where the System Integration engineer 'pulls it all together'. The areas of employment are many and varied, with the increase in 'connectivity' the employment opportunities are now across the board.

A major employer of the system integration engineer is the defense industry – a major focus area for these consulting companies; the military are driving the whole discipline of 'connectivity'. The need for information, or more usefully, 'wisdom' is an insatiable need. Different levels of information are needed by different levels of military commanders, from the broad strategic information needed by senior military commanders to the localized knowledge needed by the front line soldier.

A major measure of the information is its 'currentness', information more than a couple of minutes old is often useless, not only is information needed about what is happening now, information is needed about what is likely to happen at some point in the future. In recent years the job description of 'System Integration Engineer' has become very broad. Any system that connects to another could be defined as a system that needs integration and, therefore, a System Integration engineer. This trend is likely to continue with the growth of the Internet and the utilities that use it.

4.1.3.1 Methods of integration

Vertical Integration is process of integrating subsystems according to their functionality by creating functional entities also referred to as silos. The benefit of this method is that the integration is performed fast and with involving only the necessary vendors, therefore, this method is cheaper in short term. On the other hand, cost-of-ownership can be substantially higher than seen in the other methods, since in case of new or enhanced functionality, the only possible way to implement (scale the system) would be by implementing another silo. Reusing subsystems to create a different level of functionality is not possible.

Star Integration or also known as Spaghetti Integration is process of integration of the systems where each system is interconnected to each of the remaining subsystems. When observed from the perspective of the subsystem which has been integrated, it reminds one of a star, but when the overall diagram of the system is presented, the connections look like

spaghetti, hence, the name of this method. The cost of this method of integration can vary from the interfaces which subsystems are exporting. In case in which the subsystems are exporting vendor-specific interfaces, the integration cost can substantially rise. Time and costs needed to integrate the systems exponentially rises by adding additional subsystems. From the perspective of implementing new features, this method is preferable since it provides extreme flexibility to reuse the functionalities from existing subsystem into new system.

Horizontal Integration or Enterprise service bus is a method in which a specialized subsystem (BUS) is added to the system which is dedicated to communicate with other subsystems. This allows cutting the number of connections (interfaces) to only one per subsystem which connects directly to the BUS. The BUS is capable to translate the interface into another interface. This allows cutting the costs of integration and provides extreme flexibility. With systems integrated by this method, it is possible to completely replace one subsystem with another subsystem which provides similar functionality but exports different interfaces, all this completely transparent for the rest of the subsystems. The only required thing is to implement the new interface between the BUS and the new subsystem.

4.1.4 ENTERPRISE APPLICATION INTEGRATION

Enterprise Application Integration (EAI) is defined as the uses of software and computer systems architectural principles to integrate a set of enterprise computer applications. In today's competitive and dynamic business environment, applications such as Supply Chain Management, Customer Relationship Management, Business Intelligence and Integrated Collaboration environments have become imperative for organizations that need to maintain their competitive advantage. Enterprise Application Integration (EAI) is the process of linking these applications and others in order to realize financial and operational competitive advantages.

When different systems can't share their data effectively, they create information bottlenecks that require human intervention in the form of decision making or data entry. With a properly deployed EAI architecture, organizations are able to focus most of their efforts on their value-creating core competencies instead of focusing on work flow management.

For generations, systems have been built that have served a single purpose for a single set of users without sufficient thought to integrating these systems into larger systems and multiple applications. EAI is the solution to the unanticipated outcome of generations of development undertaken without a central vision or strategy. The demand of the enterprise is to share data and processes without having to make sweeping changes to the applications or data structures. Only by creating a method of accomplishing this integration can EAI be both functional and cost-effective. One of the challenges facing modern organizations is giving all their workers complete, transparent and real-time access to information. Many of the legacy applications still in use today were developed using arcane and proprietary technologies, thus creating information silos across departmental lines within organizations. These systems hampered seamless movement of information from one application to the other. EAI, as a discipline, aims to alleviate many of these problems, as well as create new paradigms for truly lean proactive organizations. EAI intends to transcend the simple goal of linking applications, and attempts to enable new and innovative ways of leveraging organizational knowledge to create further competitive advantages for the enterprise.

EAI is a response to decades of creating distributed monolithic, single purpose applications leveraging a hodgepodge of platforms and development approaches. EAI represents the solution to a problem that has existed since applications first moved from central processors. Undoubtedly, there are a number of instances of stovepipe systems in an enterprise, such as inventory control systems, sales automation systems, general ledger systems, and human resource systems. These systems typically were custom-built with specific needs in mind, utilizing the technology-of-the-day. Many used non-standard data storage and application development technology. There are some basic reasons for EAI in large organizations. Enterprise Application Integration has increased in importance because enterprise computing often takes the form of islands of automation. This occurs when the value of individual systems are not maximized because of the partial or full isolation. If integration is applied without following a structured EAI approach, point-to-point connections grow across an organization. Dependencies are added on an impromptu basis, resulting in a tangled mess that is difficult to maintain. This is commonly referred to as spaghetti, an allusion to the programming equivalent of spaghetti code. For example:

The number of n connections needed to have a fully meshed point-to-point connections is given by

$$\frac{n(n-1)}{2}$$

Thus, for 10 applications to be fully integrated point-to-point,

$$\frac{(10)(9)}{2}$$

or 45 point-to-point connections are needed.

However, EAI is not just about sharing data between applications; it focuses on sharing both business data and business process. Attending to EAI involves looking at the system of systems, which involves large scale inter-disciplinary problems with multiple, heterogeneous, distributed systems that are embedded in networks at multiple levels.

EAI can be used for different purposes:

a) Data (information) integration: ensuring that information in multiple systems is kept consistent. This is also known as EII (Enterprise Information Integration).

b) Process integration: linking business processes across applications.

c) Vendor independence: extracting business policies or rules from applications and implementing them in the EAI system, so that even if one of the business applications is replaced with a different vendor's application, the business rules do not have to be re-implemented.

d) Common facade: An EAI system could front-end a cluster of applications, providing a single consistent access interface to these applications and shielding users from having to learn to interact with different applications.

4.1.4.1 Integration Patterns

There are two patterns that EAI systems implement:

Mediation: Here, the EAI system acts as the go between or broker between (interface or communicating) multiple applications. Whenever an interesting event occurs in an application (e.g., new information created, new transaction completed, etc.) an integration module in the EAI system is notified. The module then propagates the changes to other relevant applications.

Federation: In this case, the EAI system acts as the overarching facade across multiple applications. All accesses from the 'outside world' to any of the applications are front-ended by the EAI system. The EAI system is configured to expose only the relevant information and interfaces of the underlying applications to the outside world, and performs all interactions with the underlying applications on behalf of the requester.

Both patterns are often used concurrently. The same EAI system could be keeping multiple applications in sync (mediation), while servicing requests from external users against these applications (federation).

Access Patterns – EAI supports both asynchronous and synchronous access patterns, the former being typical in the mediation case and the latter in the federation case.

Lifetime Patterns – An integration operation could be short-lived (e.g., keeping data in sync across two applications could be completed within a second) or long-lived (e.g., one of the steps could involve the EAI system interacting with a human work flow application for approval of a loan that takes hours or days to complete).

EAI Topologies – There are two major topologies: hub-and-spoke, and bus. Each has its own advantages and disadvantages. In the hub-and-spoke model, the EAI system is at the center (the hub), and interacts with the applications via the spokes. In the bus model, the EAI system is the bus (or is implemented as a resident module in an already existing message bus or message-oriented middleware).

4.1.4.2 Technologies

Multiple technologies are used in implementing each of the components of the EAI system:

Bus/hub: This is usually implemented by enhancing standard middleware products (application server, message bus) or implemented as a stand-alone program (i.e., does not use any middleware), acting as its own middleware.

Application connectivity: The bus/hub connects to applications through a set of adapters (also referred to as connectors). These are programs that know how to interact with an underlying business application. The adapter performs two-way communication, performing requests from the hub against the application, and notifying the hub when an event of interest occurs in the application (a new record inserted, a transaction completed, etc.). Adapters can be specific to an application (e.g., built against the application vendor's client libraries) or specific to a class of applications (e.g., can interact with any application through a standard communication protocol, such as SOAP or SMTP). The adapter could reside in the same process space as the bus/hub or execute in a remote location and interact with the hub/bus through industry standard protocols such as message queues, web services, or even use a proprietary protocol. In the Java world, standards such as JCA allow adapters to be created in a vendor-neutral manner.

Data format and transformation: To avoid every adapter having to convert data to/from every other applications' formats, EAI systems usually stipulate an application-independent (or common) data format. The EAI system usually provides a data transformation service as well to help convert between application-specific and common formats. This is done in two steps: the adapter converts information from the application's format to the bus's common format. Then, semantic transformations are applied on this (converting zip codes to city names, splitting/merging objects from one application into objects in the other applications, and so on).

Integration modules: An EAI system could be participating in multiple concurrent integration operations at any given time, each type of integration being processed by a different integration module. Integration modules subscribe to events of specific types and process notifications that they receive when these events occur. These modules are implemented in

different ways: on Java-based EAI systems, these could be web applications or EJBs or even POJOs that conform to the EAI system's specifications.

Support for transactions: When used for process integration, the EAI system also provides transactional consistency across applications by executing all integration operations across all applications in a single overarching distributed transaction (using two-phase commit protocols or compensating transactions).

4.1.4.3 Communication Architecture

Currently, there is a lot of variation of thought on what constitutes the best infrastructure, component model, and standards structure for Enterprise application integration. There seems to be consensus that four things are essential for modern enterprise application architecture:

1) There needs to be a centralized broker that handles security, access, and communication. This can be accomplished through integration servers (like the School Interoperability Framework (SIF) Zone Integration Servers) or through similar software like the Enterprise service bus (ESB) model which acts as a SOAP-oriented services manager.

2) The use of an independent data model based on a standard data structure. It appears that XML and the use of XML style sheets has become the de facto and in some cases de jure standard.

3) A connector, or agent, model where each vendor, application , or interface can build a single component that can speak natively to that application and communicate with the centralized broker.

4) A system model that defines the APIs, data flow and rules of engagement to the system such that components can be built to interface with it in a standardized way.

Although other approaches like connecting at the database or user-interface level have been explored, they have not been found to scale or be able to adjust. Individual applications can publish messages to the centralized broker and subscribe to receive certain messages from that broker. Each application requires only one connection to the broker. This central control

approach can be extremely scalable and highly evolvable. Enterprise Application Integration is related to the middleware technologies such as message-oriented middleware (MOM), and data representation technologies such as XML. Other EAI technologies involve using web services as part of service-oriented architecture as a means of integration. Enterprise Application Integration tends to be data centric. In the near future, it will come to include content integration and business processes.

4.1.5 INFRASTRUCTURE MANAGEMENT SERVICES

For an organization's information technology, infrastructure management (IM) is the management of essential operation components, such as policies, processes, equipment, data, human resources, and external contacts, for overall effectiveness. Infrastructure management is sometimes divided into categories of systems management, network management, and storage management. Infrastructure management products are available from a number of vendors including Hewlett-Packard, IBM, CA and Microsoft. According to an IDC survey *(IDC, 2003)* of 105 enterprise business and technology professionals, centralized IT operations and technology standardization are growing as the pressure to align IT with business processes increases. Yet users continue to underestimate the time frame for developing business innovation from their technology investments. To avoid expensive project failures and increase the success of managing business services, IDC believes that infrastructure management offers the greatest impact on revenue for businesses.

As IT organizations change to become more service-centric and move away from component management, they must address technology architecture changes, process adjustments, and staff training. It's clear, through the survey respondents, that these areas will be at the top of mind for most IT and business professionals during the next 5-10 years, as linking business and technology processes grows in importance and impact. To increase the success of both technology investments and new business services, IDC advises businesses to increase the communications between IT and business professionals, be flexible to necessary organizational changes, and have a vision that is supported by C-level executives and propagated throughout the various IT groups.

4.1.6 TESTING SERVICES

Besides the testing methodologies mentioned and explained earlier that are normally undertaken by the development house itself using a specialist team, automated testing services are also provided by these consulting services. Test automation is the use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions. Commonly, test automation involves automating a manual process already in place that uses a formalized testing process.

Over the past few years, tools that help programmers quickly create applications with graphical user interfaces (GUI) have dramatically improved programmer productivity. This has increased the pressure on testers, who are often perceived as bottlenecks to the delivery of software products. Testers are being asked to test more and more code in less and less time. Test automation is one way to do this, as manual testing is time consuming. As different versions of software are released, the new features will have to be tested manually time and again. But, now there are tools available that help the testers in the automation of the GUI which reduce the test time as well as the cost; other test automation tools support execution of performance tests. Many test automation tools provide record and playback features that allow users to record interactively user actions and replay it back any number of times, comparing actual results to those expected. However, reliance on these features poses major reliability and maintainability problems. Most successful automators use a software engineering approach, and as such most serious test automation is undertaken by people with development experience.

A growing trend in software development is to use testing frameworks such as the xUnit frameworks (for example, JUnit and NUnit) which allow the code to conduct unit tests to determine whether various sections of the code are acting as expected in various circumstances. Test cases describe tests that need to be run on the program to verify that the program runs as expected. All three aspects of testing can be automated. Another important aspect of test automation is the idea of partial test automation, or automating parts but not all of the software testing process. If, for example, an oracle cannot reasonably be created, or if

fully automated tests would be too difficult to maintain, then a software tools engineer can instead create testing tools to help human testers perform their jobs more efficiently. Testing tools can help automate tasks such as product installation, test data creation, GUI interaction, problem detection (consider parsing or polling agents equipped with oracles), defect logging, etc., without necessarily automating tests in an end-to-end fashion.

Test automation is expensive and it is an addition, not a replacement, to manual testing. It can be made cost-effective in the longer term though, especially in regression testing. One way to generate test cases automatically is model-based testing where a model of the system is used for test case generation, but research continues into a variety of methodologies for doing so.

4.1.7 SERVICE ORIENTED ARCHITECTURE

Service Oriented Architecture (SOA) is an architectural style that guides all aspects of creating and using business processes, packaged as services, throughout their lifecycle, as well as defining and provisioning the IT infrastructure that allows different applications to exchange data and participate in business processes regardless of the operating systems or programming languages underlying those applications. SOA represents a model in which functionality is decomposed into small, distinct units (services), which can be distributed over a network and can be combined together and reused to create business applications. These services communicate with each other by passing data from one service to another, or by coordinating an activity between one or more services. It is often seen as an evolution of distributed computing and modular programming.

Companies have long sought to integrate existing systems in order to implement information technology (IT) support for business processes that cover all present and prospective systems requirements needed to run the business end-to-end. A variety of designs can be used to this end, ranging from rigid point-to-point electronic data interchange (EDI) interactions to Web auctions. By updating older technologies, such as Internet-enabling EDI-based systems, companies can make their IT systems available to internal or external customers; but the resulting systems have not proven to be flexible enough to meet business demands. A flexible, standardized architecture is therefore required to better support the connection of various

applications and the sharing of data. SOA is one such architecture. It unifies business processes by structuring large applications as an ad-hoc collection of smaller modules called services. These applications can be used by different groups of people both inside and outside the company, and new applications built from a mix of services from the global pool exhibit greater flexibility and uniformity. One should not, for example, have to provide redundantly the same personal information to open an online checking, savings or IRA account, and further, the interfaces one interacts with should have the same look and feel and use the same level and type of input data validation. Building all applications from the same pool of services makes achieving this goal much easier and more deployable to affiliate companies. An example of this might be interacting with a rental car company's reservation system even though you are doing so from an airline's reservation system.

SOAs build applications out of software services. Services are relatively large, intrinsically unassociated units of functionality, which have no calls to each other embedded in them. They typically implement functionalities most humans would recognize as a service, such as filling out an online application for an account, viewing an online bank statement, or placing an online book or airline ticket order. Instead of services embedding calls to each other in their source code, protocols are defined which describe how one or more services can talk to each other. This architecture then relies on a business process expert to link and sequence services, in a process known as orchestration, to meet a new or existing business system requirement.

Relative to earlier attempts to promote software reuse via modularity of functions, or by use of predefined groups of functions known as classes, SOA's atomic level objects are 100 to 1,000 times larger, and are associated by an application designer or engineer using orchestration. In the process of orchestration, relatively large chunks of software functionality (services) are associated in a non-hierarchical arrangement (in contrast to a class's hierarchies) by a software engineer, or process engineer, using a special software tool which contains an exhaustive list of all of the services, their characteristics, and a means to record the designer's choices which the designer can manage and the software system can consume and use at run-time.

Underlying and enabling all of this is metadata which is sufficient to describe not only the characteristics of these services, but also the data that drives them. XML has been used

extensively in SOA to create data which is wrapped in a nearly exhaustive description container. Analogously, the services themselves are typically described by WSDL, and communications protocols by SOAP. Whether these description languages are the best possible for the job, and whether they will remain the favorites going forwards, is at present an open question. What is certain is that SOA is utterly dependent on data and services that are described using some implementation of metadata which meets two criteria. The metadata must be in a form which software systems can consume to dynamically configure to maintain coherence and integrity, and in a form which system designers can understand and use to manage that metadata.

The goal of SOA is to allow fairly large chunks of functionality to be strung together to form ad-hoc applications which are built almost entirely from existing software services. The larger the chunks, the fewer the interface points required to implement any given set of functionality; however, very large chunks of functionality may not be granular enough to be easily reused. Each interface brings with it some amount of processing overhead, so there is a performance consideration in choosing the granularity of services. The great promise of SOA is that the marginal cost of creating the n-th application is zero, as all of the software required already exists to satisfy the requirements of other applications. Only orchestration is required to produce a new application.

The key is that there are no interactions between the chunks specified within the chunks themselves. Instead, the interaction of services (all of whom are unassociated peers) is specified by humans in a relatively ad-hoc way with the intent driven by newly emergent business requirements. Thus the need for services to be much larger units of functionality than traditional functions or classes, lest the sheer complexity of thousands of such granular objects overwhelm the application designer. The services themselves are developed using traditional languages like Java, C#, C++, C or COBOL.

SOA services are loosely coupled, in contrast to the functions a linker binds together to form an executable, a dynamically linked library, or an assembly. SOA services also run in "safe" wrappers such as Java or .NET, which manage memory allocation and reclamation, allow ad-hoc and late binding, and provide some degree of indeterminate data typing.

Increasing numbers of third-party software companies are offering software services for a fee. In the future, SOA systems may consist of such third-party services combined with others created in-house. This has the potential to spread costs over many customers, and customer uses, and promotes standardization both in and across industries. In particular, the travel industry now has a well-defined and documented set of both services and data, sufficient to allow any reasonably competent software engineer to create travel agency software using entirely off-the-shelf software services. Other industries, such as the finance industry, are also making significant progress in this direction.

SOA is an architecture that relies on service-orientation as its fundamental design principle. In an SOA environment independent services can be accessed without knowledge of their underlying platform implementation *(Tuggle, 2003)*.

Base requirements for an SOA – In order to efficiently use an SOA, one must meet the following requirements:

a) Interoperability between different systems and programming languages provides the basis for integration between applications on different platforms through a communication protocol. One example of such communication is based on the concept of messages. Using messages across defined message channels decreases the complexity of the end application thereby allowing the developer of the application to focus on true application functionality instead of the intricate needs of a communication protocol.

b) Desire to create a federation of resources. Establish and maintain data flow to a federated data warehouse. This allows new functionality developed to reference a common business format for each data element.

Web services can be used to implement a service-oriented architecture. A major focus of Web services is to make functional building blocks accessible over standard Internet protocols that are independent from platforms and programming languages. These services can be new applications or just wrapped around existing legacy systems to make them network-enabled.

Each SOA building block can play one or more of three roles:

1. Service provider – The service provider creates a Web service and possibly publishes its interface and access information to the service registry. Each provider must decide which services to expose, how to make trade-offs between security and easy availability, how to price the services, or, if they are free, how to exploit them for other value. The provider also has to decide what category the service should be listed in for a given broker service and what sort of trading partner agreements are required to use the service.

2. Service broker – The service broker, also known as service registry, is responsible for making the Web service interface and implementation access information available to any potential service requestor. The implementer of the broker decides about the scope of the broker. Public brokers are available through the Internet, while private brokers are only accessible to a limited audience, for example, users of a company intranet. Furthermore, the amount of the offered information has to be decided. Some brokers specialize in many listings. Others offer high levels of trust in the listed services. Some cover a broad landscape of services and others focus within an industry. There are also brokers that catalog other brokers. Depending on the business model, brokers can attempt to maximize look-up requests, number of listings or accuracy of the listings. The Universal Description Discovery and Integration (UDDI) specification defines a way to publish and discover information about Web services.

3. Service requestor – The service requestor or Web service client locates entries in the broker registry using various find operations and then binds to the service provider in order to invoke one of its Web services.

Architecture is not tied to a specific technology *(Erl, 2005)*. It may be implemented using a wide range of technologies, including SOAP, RPC, DCOM, CORBA, Web Services or WCF. SOA can be implemented using one or more of these protocols and, for example, might use a file system mechanism to communicate data conforming to a defined interface specification between processes conforming to the SOA concept.

The key is independent services with defined interfaces that can be called to perform their tasks in a standard way, without the service having foreknowledge of the calling application, and without the application having or needing knowledge of how the service actually performs its tasks.

SOA can also be regarded as a style of information systems architecture that enables the creation of applications that are built by combining loosely coupled and interoperable services. These services inter-operate based on a formal definition (or contract, e.g., WSDL) that is independent of the underlying platform and programming language. The interface definition hides the implementation of the language-specific service. SOA-based systems can therefore be independent of development technologies and platforms (such as Java, .NET etc). Services written in C# running on .NET platforms and services written in Java running on Java EE platforms, for example, can both be consumed by a common composite application (or client). Applications running on either platform can also consume services running on the other as Web services, which facilitates reuse. Many COBOL legacy systems can also be wrapped by a managed environment and presented as a software service. This has allowed the useful life of many core legacy systems to be extended indefinitely no matter what language they were originally written in.

SOA can support integration and consolidation activities within complex enterprise systems, but SOA does not specify or provide a methodology or framework for documenting capabilities or services. High-level languages such as BPEL and specifications such as WS-CDL and WS-Coordination extend the service concept by providing a method of defining and supporting orchestration of fine grained services into more coarse-grained business services, which in turn can be incorporated into workflows and business processes implemented in composite applications or portals. The use of Service component architecture (SCA) to implement SOA is a current area of research.

Enterprise architects believe that SOA can help businesses respond more quickly and cost-effectively to changing market conditions [7]. This style of architecture promotes reuse at the macro (service) level rather than micro (classes) level. It can also simplify interconnection to - and usage of existing IT (legacy) assets.

In some respects, SOA can be considered an architectural evolution rather than a revolution and captures many of the best practices of previous software architectures. In communications systems, for example, there has been little development of solutions that use truly static bindings to talk to other equipment in the network. By formally embracing an SOA approach, such systems are better positioned to stress the importance of well-defined, highly inter-operable interfaces.

It may be asked whether SOA is just a revival of modular programming (1970s), event-oriented design (1980s) or interface/component-based design (1990s). SOA promotes the goal of separating users (consumers) from the service implementations. Services can therefore be run on various distributed platforms and be accessed across networks. This can also maximize reuse of services. SAP is doing a lot of work in this area and have designed their current ERP release *(mySAP 2005)* around SOA.

The following guiding principles define the ground rules for development, maintenance, and usage of SOA *(Balzer, 2004)*

- Reuse, granularity, modularity, composability, componentization, and interoperability
- Compliance to standards (both common and industry-specific)
- Services identification and categorization, provisioning and delivery, and monitoring and tracking

The following specific architectural principles for design and service definition focus on specific themes that influence the intrinsic behavior of a system and the style of its design:

- Service Encapsulation - A lot of existing web-services are consolidated to be used under the SOA Architecture. Many a times, such services have not been planned to be under SOA.
- Service Loose coupling - Services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other

- Service contract - Services adhere to a communications agreement, as defined collectively by one or more service description documents

- Service abstraction - Beyond what is described in the service contract, services hide logic from the outside world

- Service reusability - Logic is divided into services with the intention of promoting reuse

- Service composability - Collections of services can be coordinated and assembled to form composite services

- Service autonomy – Services have control over the logic they encapsulate

- Service optimization – All else equal, high-quality services are generally considered preferable to low-quality ones

- Service discoverability – Services are designed to be outwardly descriptive so that they can be found and assessed via available discovery mechanisms

In addition, the following factors should also be taken into account when defining an SOA implementation:

- SOA Reference Architecture covers the SOA Reference Architecture, which provides a worked design of an enterprise-wide SOA implementation with detailed architecture diagrams, component descriptions, detailed requirements, design patterns, opinions about standards, patterns on regulation compliance, standards templates etc.

- Life cycle management SOA Practitioners Guide Part 3: Introduction to Services Lifecycle introduces the Services Lifecycle and provides a detailed process for services management though the service lifecycle, from inception through to retirement or repurposing of the services. It also contains an appendix that

includes organization and governance best practices, templates, comments on key SOA standards, and recommended links for more information.

- Efficient use of system resources
- Service maturity and performance
- EAI Enterprise Application Integration

There are also some challenges SOA faces. One obvious and common challenge faced is managing services metadata. SOA-based environments can include many services which exchange messages to perform tasks. Depending on the design, a single application may generate millions of messages. Managing and providing information on how services interact is a complicated task. Another challenge is providing appropriate levels of security. Security model built into an application may no longer be appropriate when the capabilities of the application are exposed as services that can be used by other applications. That is, application-managed security is not the right model for securing services. A number of new technologies and standards are emerging to provide more appropriate models for security in SOA. As SOA and the WS-* specifications are constantly being expanded, updated and refined, there is a shortage of skilled people to work on SOA based systems, including the integration of services and construction of services infrastructure.

Interoperability is another important aspect in the SOA implementations. The WS-I organization has developed Basic Profile (BP) and Basic Security Profile (BSP) to enforce compatibility. Testing tools have been designed by WS-I to help assess whether web services are conformant with WS-I profile guidelines. Additionally, another Charter has been established to work on the Reliable Secure Profile. There is significant vendor hype concerning SOA that can create expectations that may not be fulfilled. Product stacks are still evolving as early adopters test the development and runtime products with real world problems. SOA does not guarantee reduced IT costs, improved systems agility or faster time to market. Successful SOA implementations may realize some or all of these benefits depending on the quality and relevance of the system architecture and design *(Computerworld, July 2006).*

SOA has also been criticized on several fronts. Some criticisms of SOA are based on the assumption that SOA is just another term for Web Services. For example, some critics claim SOA results in the addition of XML layers introducing XML parsing and composition. In the absence of native or binary forms of Remote Procedure Call (RPC) applications could run slower and require more processing power, increasing costs. Most implementations do incur these overheads, but SOA can be implemented using technologies (for example, Java Business Integration (JBI)) which do not depend on remote procedure calls or translation through XML. At the same time, there are emerging, open-source XML parsing technologies, such as VTD-XML, and various XML-compatible binary formats that promise to significantly improve the SOA performance.

Stateful services require both the consumer and the provider to share the same consumer-specific context, which is either included in or referenced by messages exchanged between the provider and the consumer. The drawback of this constraint is that it could reduce the overall scalability of the service provider because it might need to remember the shared context for each consumer. It also increases the coupling between a service provider and a consumer and makes switching service providers more difficult. Another concern is that WS-* standards and products are still evolving (e.g., transaction, security), and SOA can thus introduce new risks unless properly managed and estimated with additional budget and contingency for additional proof of concept work. An informal survey by Network Computing placed SOA as the most despised buzzword (November 2006). Some critics feel SOA is merely an obvious evolution of currently well-deployed architectures (open interfaces, etc).

SOA architecture is the first stage of representing the system components that interconnect for the benefit of the business. At this level a SOA is just an evolution of an existing architecture and business functions. SOAs are normally associated with interconnecting back end transactional systems that are accessed via web services. The real issue with any IT "architecture" is how one defines the information management model and operations around it that deal with information privacy, reflect the business's products and services, enable services to be delivered to the customers, allow for self care, preferences and entitlements and at the same time embrace identity management and agility. On this last point, system modification (agility) is a critical issue which is normally omitted from IT system design. Many

systems, including SOAs, hard code the operations, goods and services of the organization thus restricting their online service and business agility in the global market place. Adopting SOAs is therefore just the first (diagrammatic) step in defining a real business system. The next step in the design process is the definition of a Service Delivery Platform (SDP) and its implementation. It is in the SDP design phase where one defines the business information models, identity management, products, content, devices, and the end user service characteristics, as well as how agile the system is so that it can deal with the evolution of the business and its customers.

One area where SOA has been gaining ground is in its power as a mechanism for defining business services *(Jones, 2006)* and operating models and thus providing a structure for IT to deliver against the actual business requirements and adapt in a similar way to the business. The purpose of using SOA as a business mapping tool is to ensure that the services created adequately represent the business view and are not just what technologists think the business services should be. At the heart of SOA planning is the process of defining architectures for the use of information in support of the business, and the plan for implementing those architectures *(Enterprise Architecture Planning by Steven Spewak and Steven Hill).* Enterprise Business Architecture should always represent the highest and most dominant architecture. Every service should be created with the intent to bring value to the business in some way and must be traceable back to the business architecture.

Within this area, SOMA (Service-Oriented Modeling and Architecture) was announced by IBM as the first publicly announced SOA-related methodology in 2004. Since then, efforts have been made to move towards greater standardization and the involvement of business objectives, particularly within the OASIS standards group and specifically the SOA Adoption Blueprints group. All of these approaches take a fundamentally structured approach to SOA, focusing on the Services and Architecture elements and leaving implementation to the more technically focused standards.

4.1.8 COMPLIANCE

This is one of the current emerging areas worldwide that is getting more widely adopted as well as matured in terms of standards and specifications. Common sets of compliance worldwide as shown in Table 4.1 include:

Table 4.1 - Common sets of compliance worldwide

Global & Emerging Regulations
▪ USA: SOX
▪ Japan: J-SOX
▪ UK: Combined Code
▪ France: LSF
▪ Italy: 231 & 262
▪ Sweden: Corporate Code
▪ Switzerland: Swiss Code
▪ EU: 4th, 7th & 8th Directives
▪ Brazil: Governanca Corportiva
▪ Russia: Order No. 04-1245
▪ India: Clause 49
▪ China: SASAC Directive
▪ Australia: CLERP 9
▪ Global: BASEL II
▪ etc.

Reference:
Ernst & Young LLP, 2006, Leveraging Value from Internal Controls, Ernst & Young EYGM Limited,

There are a myriad of technologies and vendors to support an organization's quest for achieving compliance. These range from products in data storage to security and identity based access. It is the domain of these consulting companies to select an appropriate product (or combination of products) and technology to enable their clients to get results.

To recapitulate, in chapter three and four, I have explained and examined IT tools, technologies and methodologies that the consulting companies have developed practices in the different areas of Information Technology. In the following chapter, I cover the target clientele of these companies in the Energy and allied sectors, and examine, through a series of cases the benefits that have accrued to these companies' clients by virtue of their efforts and proper implementation of the IT tools and methodologies.