Unprecedented Cost Reduction and Customer Satisfaction through Stable, Precise, and Concise Requirement Model

Waterfall lifecycle specifies all requirements upfront and practices change control management. It minimizes rework but maximizes misalignment between business and software as a study showed that 45% of implemented features were never used and only 20% were often used. Agile lifecycle uses short value-based iterations to implement growing requirement needs in the process of coding. It minimizes business and software misalignment but maximizes rework for its continuous architectural change to compensate the discrepancies between existing and new code. This paper introduces a new concept to requirement modeling to minimize rework and maximize business and software alignment all at the same time.

Keywords: enterprise software, business process, business rule, user requirement, and systems thinking

Traditionally, software development processes have been requirement-driven: an attempt is made to provide requirement definitions and then to implement those requirements. Therefore, the success or failure of requirements determines the success or failure of software projects. In real world however, virtually every major software program suffers from severe difficulties in requirement specification. Many studies cite requirement management issues as a primary cause of software project failure. The Standish Group's CHAOS Reports from 1994 and 1997 established that the most significant contributors to project failure relate to requirement.

Requirement specification is commonly defined as "shall statements," what the software should do. Analysts elicit requirements from each user and record the requirements in documents that run to hundreds, sometimes thousands, of pages. This format of representing requirement does not convey what system is truly needed. It focuses on the system, what the system shall do, the solution, rather than on the understanding of the business, what the business should do. There are three distinct problems.

1. *Subjective uncertainty* - It is well known that users do not know what requirements are. When pressured for answers, users are forced to offer their best guesses. Hence, requirements are built on shifting grounds - personal opinions. There is no subjective certainty from users when they create requirements. This uncertainty passes on to analysts and developers.
2. *Artifact incoherence* - The format of requirement spec does not facilitate logically the action in the next step. If use case modeling is next step, it does not inform how use cases are identified and their relations. Analysts will have to model use cases based on their own imaginations with little use of requirement specs. Because artifacts are not built upon each other from one form to the other, analysts and designers mostly build their own artifacts independently by talking to users whenever needed. Resultantly, these artifacts seldom express the system consistently in different forms from different perspectives. Incoherence leads to waste of resources on useless work (artifacts created but never used) and rework (change of existing artifacts for missing information).
3. *Tight coupling between business and software* - User centric approach focuses exclusively on domain-specific solutions that are tightly coupled with, often partial understood or misunderstood, domains of businesses. Accordingly, software is becoming more customized and correspondingly less generic. While some end users may be able to request features that closely fit their business processes, it's likely that most of us end up with a poor fit between software solution and business needs. Because of the narrow coverage of the business by the proposed solution and inflexibility of change for each, there seems to be the constant need for new applications to accommodate changing needs. The end result is massive cross-over duplication of development of software that tries to implement code as well as business logic. These duplicated development efforts create siloed applications that can't work together for tight coupling of software and business.

The solution to the above three problems can't be found in the prevailing methodologies currently seen in the market. This is because software engineering has been historically emulating traditional engineering design approach to attack problems. That is, software is like machines and we can learn how to build software by talking to its users. This paper introduces a new requirement approach based on a shift in thinking that corresponds to insights gleaned from living systems view and systems thinking. Enterprise software systems are not machines but purposeful living systems. Using the concept of systems and systems thinking, we'd be able to

model the business effectively and build software that direct supports the business. Accordingly we achieve the goal of subjective certainty, artifact coherence, and decoupling between business and software.

The Concept of Systems and Systems Thinking

A system is defined as a set of two or more elements that satisfy the following three conditions:

1. The behaviour of each element has an effect on the behaviour of the whole.
2. How an element's behaviour affects the whole depends on the behaviour of at least one other element.
3. Elements can be grouped as subsets that act as elements of the whole

A system therefore is a whole that cannot be divided into independent parts. The property of the system, the defining property is the property of the whole that none of the parts has. The property of an automobile is to move people from one place to another. No part in an automobile can carry you from one place to another. An engine separated from an automobile looses its capacity to move. Only the automobile can move you from one place to another. Assembling Buick's brake, BMW's transmission, and Royce Roller's engine together does not give you a best car. In fact, it is not even a car. The parts do not match. The automobile is the product of the interaction, not the sum, of its parts. A function of a company is to provide value to its customers none of its employees and software systems can do it alone.

A system differentiates itself from its environment by having a boundary. The boundary of a system divides those elements as within the boundary to be part of the system from those elements in the environment of the system. It is an easy matter to redraw the boundary of a system on paper at a very early stage of development. However, as a project progresses the boundary becomes embedded in the design concept, investment is made, and it becomes progressively more difficult to alter the position of the boundary. Placement of a boundary reflects the perspective of the system's designer and is vitally important to the success of the system.

How the boundary of a system is drawn depends on the purpose to which the system is designed to serve. Any system relates to three levels of purposes: purpose of the parts, of the system itself, and of the objects external to the system. An automobile, for example, having no purpose of its own and of its parts, serves the purpose of its users. Therefore the boundary of an automobile excludes its users as part of the automobile. The same is with tax software whose boundary excludes its users, taxpayers as elements within the system. Systems, that have no purpose of their own and of their parts but serve purposes of something external to them, are deterministic systems. Examples include mechanical systems, Microsoft Word, and game software. Social systems are purposeful and so are their parts, the human beings. The purpose of a company may be to make profits and distribute wealth by serving the purpose of its customers through products and services. The employees produce products and services to fulfill the purpose of the company not the purpose of the customers. Therefore, the boundary of the company divides its employees and tools they use as part of the company from its customers and partners in the environment. Anything serving the purpose of the system belongs to the part of the system and anything the system serves is in the environment.

A system that could carry out its function in every environment would be completely independent of its environment and therefore be closed. Deterministic systems are closed. A system that requires certain environmental conditions in order to carry out its defining function is an open system. Social systems are open. The environment of a system consists of those things that can affect the properties and performance of that system, but over which it has no control. That part of its environment that a system can influence, but not control, is said to be *transactional*. Customers and suppliers, for example, are parts of a corporation's transactional environment. That parts of a system's environment that can neither be influenced nor controlled is said to be *contextual*, for example, the weather and government regulation.

It is important to differentiate analytical thinking from synthetic thinking to study systems. In analytical thinking, the thing to be explained is treated as a whole to be taken apart. In synthetic thinking, the thing to be explained is treated as a part of containing whole. The number of seats of an automobile is not reducible to the parts but can only be explained by the context of its use through synthesis. Analysis looks into things and synthesis looks out of things. Synthesizing a social system is to formulate its business needs, processes of delivering products and services to its customers. Analysing a social system is to formulate its members' or users' needs, processes of producing products and services. The interactions between the parts within the system are explained not only in terms of internal relationships but also of the external couplings with the environment. Analytical thinking and synthetic thinking can be considered separately but are complementary and combined in new way called systems thinking. In systems thinking, our understanding increases the larger system we comprehend while our knowledge increases the smaller elements we comprehend. Everything we need to understand depends

on the larger system. We cannot understand a university unless we understand the containing system, the education system. In analytical thinking, the understanding is from the parts to the whole. If an automobile is broken, we take it apart to see what part is not working. In systems thinking, the parts are not intrinsic to their own properties but can only be understood in the context of the whole. Thus systems-thinking is context, or environmental, thinking.

Define Enterprise Software System

Many believe that working software is the only deliverable that matters and everything else can be without. We can know what software to build by talking to its users. It means that the boundary of enterprise software systems excludes the users and includes only software code. It is true that the software we build should support the users and we can learn from the interactions between users and software. However, it is even more important that enterprise software systems support the mission for which they are built. For example, the system should provide value to the business that uses it and to its customers. Therefore enterprise software systems provide services and products to other pasts of the enterprise, internal customers, as well as external customers and/or partners. The internal and external customers constitute the transactional environment of the enterprise software system. Both users and working software of the enterprise system perform in tandem to create value to the enterprise. Therefore the boundary of enterprise software systems can be drawn to include working software and its users as parts of the system and exclude internal and external customers as elements in the transactional environment. Figure 1 shows a simplified diagram of an enterprise software system.

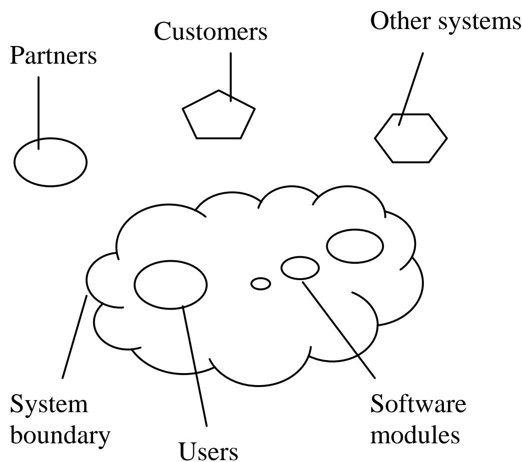

Figure 1. Simplified Enterprise Software System

Enterprise software systems are open systems because they can only carry out their functions within their enterprise. Users, as parts of the system, do not know what the system should do. Neither do they know what software components, other parts of the systems, should do. According to synthetic thinking, the understanding lies in the outside, the transactional environment of the system. Until it is understood what the system shall do as a whole can we know what its parts, the users and software components, should do by means of analytical thinking. Requirements of an enterprise software system, or any system in general, what the system shall do, comes not from inside, such as users, but from outside, its customers, partners, and other systems in the environment.

The requirement of any system, the use of the system or what the system should do, can only be understood by means of synthetic thinking. Frequent user requirement change in today's world is due to trying to study requirements through analytical thinking without synthetic thinking first. We can't understand the number of seats needed in an automobile by looking at engines and transmissions. Systems thinking requires clear defined systems boundary. A misplaced boundary would mean certain failure of a requirement definition that, in turn, leads to project failure. The boundary issue is hardly addressed by analysts today because of machine worldview of making software. By default, software systems, like machines, are closed and analysts collect requirements from users because the users are elements in the environment where the machine is used. It works for deterministic systems but may fail miserably on large-scale enterprise software systems.

Requirement Model

The totality of requirement of an enterprise software system is described as a two-level requirement model described in Figure 2. The lower level is business requirements in the form of business model and the higher level is user requirements in the form of user model. Business requirements define the problem in the business domain while user requirements describe the solution in the functional domain. Therefore user requirements realize business requirements. The user model realizes the business model by adding into it technologies and users. A business model describes interactions between the system (organizational units not users) and its transactional environment about how business value is delivered to its customers. A user model describes interactions between users and software subsystems about how business value is created. From the environment point of view, the enterprise software system is a black box. The user model goes into the black box and

describes sub-functions performed by users and software subsystems.

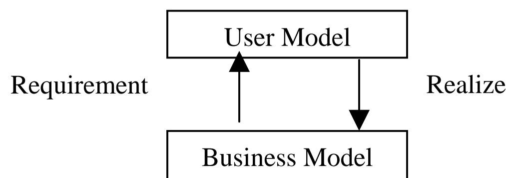

Figure 2. Requirement model

This separation of concerns between business and user requirements makes it possible and convenient to model business to our satisfaction and completion before creating the user model. The business model is independent of the user model much like physics being independent of biology. Biology relies on, but will not alter, physics in its operation. We could and should complete the study of physics before biology. With a good understanding of the business, it becomes possible to understand and model user requirements with accuracy and precision by direct translating business model into user model. In so doing, we have a requirement model that is coherent. Coherence here means that all elements in the user model are explained by elements in the business model and all elements in the business model explain elements in the user model.

Business model describes business logic while user model describes application logic. Separating business logic out of application logic allows business-oriented people to have their concerns and ideas as the basic fabric of process design rather than having a process model being imposed by outside experts. When people in the system can incorporate their individual and collective values and ideas and exercise their autonomy and responsibility for their participation and contribution, we have a business model that is authentic and sustainable because people, as the designer themselves, have the commitment and understanding.

The problems of requirement difficulty, less fit between software and business, massive cross-over duplication of development, high cost on maintenance and change can be greatly reduced when we take the business process logic out of the application code. In doing so, software solution becomes simpler and more generic. When business processes are formally modeled, the enterprise can use these business process models as an essential mechanism to communicate the business needs to the IT realm. The business and IT can significantly bridge the communication chasm by using well-articulated business models that create a link between what the business needs and what IT implements and delivers.

People may argue that business requirements will change and a stable business model can't be reached therefore it is impossible to model user requirements after business model is completed. Or, users did not fully understand what the software system ought to do until the system was almost completed. As projects proceeded and users and the developers themselves could see what the system would look like and thus came to understand the real needs better, a wealth of change would be suggested. This view is from a perspective of enterprise software being a machine. People with those arguments derived from their productive careers have committed them a fixed mind-set, the way to see problems. This commitment makes a particular perceptual blindness and rigidity to the perceptions of the world, blind to anomalies that do not fit and rigid to a single perspective. A change of perspective would mean a change in the list of problems based on the same data of experience. Ross Ashhy defines a system as "a set of variables sufficiently isolated to stay [constant] long enough for us to discuss it." A well-established enterprise is stable in terms of its competencies, type of customers, products and services. We can model these into abstract representations independent of users and the use of technologies. Even though they change, the lifecycle of such changes should be much longer than the lifecycle of software projects. If a stable business model can't be reached in timely manner, it may indicate that it is time to abort the software project for a different focus.

Model Business Requirement

Business requirements describe business processes and business rules. A business process is a structured, measured set of activities designed to achieve a defined business outcome such as delivering a product or service to a customer. Business processes have *two important characteristics*: (i) They have customers (internal or external); (ii) They go across organizational boundaries, i.e., across or between organizational subunits. Processes are generally described in terms of beginnings, end points, and in-between activities. They begin and end with customers to deliver a product or service. They are what the business, not individuals, do. Hence, a business model contains no users and no technologies. A business process is broken down into sub-processes and tasks. A sub-process is defined in the same way as a process. Tasks are the smallest units of work done by the business.

When discussing business processes, it is important to differentiate process type from process instance. A process type is a class; and it contains no software systems and no users. A process instance is an occurrence of that class and realizes the process type by adding particular customers and users in the use of software subsystems to be designed. Process type is something in general like:

- Sales process
- Processing insurance claims process

Process instance is a particular process with when, what, who and where etc, like:

- Processing a computer purchase from a particular customer at a particular time
- Processing insurance claim #1234

A business model consists of the total number of identified business process types along with business rules. Therefore, a business model is an abstraction that reduces the size of data to be processed and is a form of reductionism. It works by letting a few stands for the many. The total number of business process types within a business organization can be enumerated by the possible combinations of different kinds of products and services the business organization provides to different kinds of customers.

A business model may be modeled to satisfy following criteria:

1. Business processes are loosely coupled. Interactions between processes are asynchronous by way of message exchanges. No direct interactions between activities occur across processes.
2. Business processes may have preconditions that maybe post-conditions of other processes.
3. Sub-processes and tasks are candidates for reuse across the enterprise.

The common sense rule is to remove some degree of freedom. The business rule is to remove some degree of freedom of, or constrain, business process instances along two dimensions: people (not organization units) and objects. Constraint on the behavior of people is *operative rule*. Constraint on objects is *structural rule*. Structural rule is built-in (i.e. "structural", "by definition"). For operative rule, people may still potentially violate the rule – hence appropriate enforcement and discretion must be applied. A business rule is a simple statement that expresses the rule. Business rules dictate how the instances of a given process type should be run. Each process type is associated with a number of business rules. All such rules and the process type constitute a process definition. From a traceability perspective, business rules are traced to the steps of use cases, the instances of business process types.

Model User Requirement

A user model describes the total instances of all business process types. Hence, a user model is a direct translation from a business model. Process occurrences involve particular actors (users) in the use of the software subsystems under development. For each process type, there could be multiple process instances for the use of different technical tools such as registering by phone or by email. One process occurrence may consist of several use cases at different levels. Business tasks in business model are realized as atomic use cases. Business processes and sub-processes are realized as composite use cases that contain both atomic and composite use cases. A complete user model contains use cases, graphic user interface prototype, nonfunctional requirements and logical data model.

To achieve coherence criteria of requirement model, all tasks and processes in a business model should be realized as use cases in a user model. All use cases correspond to elements in the business model. Complete two-way traceability between two models ensures consistency, integrity and quality of the entire requirement model. Because the business model is modeled as loosely coupled between processes, so is the user model. Separation of concerns between business and user models and loosely coupled business processes will facilitate change in that changes are more of addition than modification. Modeling business requirements before modeling user requirements provides further benefits:

- Customer satisfaction – Instead of talking about what the software shall do that customers do not know, we instead talk about what the business is that customers do know about. This will increase trust and mutual understanding.
- Higher quality – The requirement model is coherent; therefore, requirements are complete, precise and concise and the system is expressed consistently at different levels in different forms.
- Greater IT and business alignment – This is possible because the proposed method first models the business as the foundation and the coherence between different models ensures business and IT alignment.
- Greater cost reduction –The coherent requirement model ensures doing the right thing throughout and hence minimizes if not eliminates rework, and reduces overhead cost (e.g. testing, quality assurance etc.)
- Effective project management – Stable and precise requirements eliminate unplanned rework and surprises. Accordingly schedule, budget and scope are much more deterministic and manageable.

Preventive Evaluation

The high waste resulting from failed and challenged projects in the software industry; especially that associated with large-scale systems failures indicates that the system of beliefs that supports thoughts about systems design is grossly underdeveloped and underconceptualized. Correcting the underconceptualized base of system thinking is like curing alcoholism. Unless the admission is made that the alcoholic needs treatment, treatment is unlikely even to be started, much less to be effective. Certain design aspects make underconceptualization most rampant and therefore grossly undermine the effectiveness of finding design solutions. They are below:

1. Underconceptualized boundaries – Setting narrow boundaries is probably the most damaging specific shortcoming in systems design. Instances are:
   - Failure to transcend the existing system or the existing setting of the design situation. It locks designers within the boundaries of the existing system where much time is spent in analyzing what's wrong. It may lead to some improvement but will never lead to a new design.
   - Lack of broad definition of the space of the systemic environment will result in a lack of compatibility between future systems and will result in the possibility of not having an adequate resource base for supporting future systems.
   - The narrow definition of time boundaries, the "hurry to show results" syndrome, is seen too frequently in design situations. This is one of the greatest sources of having "cost regret" in design. Rather than narrowing or restricting, designers should keep boundaries open and push them out as far as possible. They should "paint the largest possible picture on the largest possible canvas."
2. Underconceptualizing by "shifting down" – shifting from the whole system level down to the level of the subsystems and focusing on designing around lower-level objectives. It is a tempting proposition, particularly if the characteristics of a subsystem are well-known and easy to deal with. Optimizing the parts does not prevent the bankruptcy of the whole. Starting with the user requirements for enterprise software is a very typical general example of "shifting down".
3. Underconceptualization of perceptions, beliefs, and values – An extremely high price is paid if designers avoid issues related to perceptions, beliefs, philosophies, and values. These issues are often dismissed as being "personal" or "non practical" leading to unnecessary discussions that prolong the process. Nothing can be further from the truth. A dismissal of such issues will cause a paralysis later on in the design process. The failure to create a common frame of thinking, shared worldviews; and values as basis for making decisions is dangerously counterproductive. It will lead, in the course of design, to situations which designers are incapable of forging collective decisions. They will find themselves entangled in constant arguments, the reason being that they failed to deal with underlying assumptions and the underlying issues of their design decisions. Even more important, designers cannot transform an existing system into a new and different future system without transforming themselves. They must transcend their old way of thinking and leave old mind-sets behind. They must conceptualize a new paradigm, develop a shared way of thinking, and acquire new perspectives and a new worldview.

Conclusion

Requirement difficulties manifest themselves in almost all enterprise software development efforts and cause a huge waste of resources on the resulting failed and challenged projects as well as intellectual rework. The primary cause of such difficulties is underconceptualization by placing users outside of the boundary of enterprise software systems. This underconceptualization of shifting the whole system down to sub-systems misses true requirement of the enterprise software system. Users and software work together in tandem to create value to the business; hence are two parts of the same whole. The parts contain no knowledge of the whole and of other parts. Through synthetic thinking, the requirement of the whole can be created as business model from which a user model can be derived. The two models are hierarchically organized into requirement model that represent requirements in its entirety. The requirement model lays a solid foundation from which development proceeds. Accordingly it minimizes rework and maximizes business and IT alignment all at the same time.

Dr. Zhu was an adjunct professor at Virginia Commonwealth University between 2001 and 2004. Currently he is the president of UCSoft at Alexandria Virginia, a software process research and consulting firm that helps software companies transform their capabilities and organizational effectiveness. He can be reached at jerry.zhu@ucsoft.biz.