Customer Oriented Software Development (COSD)

Understanding the Power of Profesy™

Dr. Sofia Passova
Chief Scientist and Technology Officer
SOFEA INC.

s o f e a

bridging the gap between business & IT

CUSTOMER ORIENTED SOFTWARE DEVELOPMENT

This article considers the state of contemporary software development processes, by analyzing the major problems and their core reasons. It explores how modern methods such as CMMI and Agile Development attempt to solve these problems and why they are failing to do so. The article also provides an in-depth theoretical foundation for the necessity of a major paradigm shift in software development. A fundamentally new approach - Customer Oriented Software Development (COSD) - is proposed.

COSD changes the entire software development process in ways that will result in dramatic improvements in overall efficiency and quality. The recent commercialization of Sofea Inc.'s Profesy™ suite of tools enables, for the first time, the automated application of COSD methodology in sophisticated software development environments.

1. Software Development Today

For anyone involved, it comes as no surprise to say that software development is a time consuming, expensive process, often yielding results of disappointing quality. Today's software projects are typically plagued with time and cost overruns, producing deliverables that fall short of customer needs.

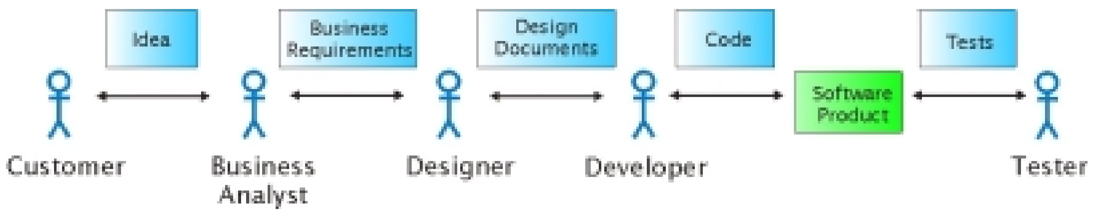
As a result of increasing awareness of the problems associated with software development and their extravagant cost to business, countless recent initiatives have focused on process improvements. Despite these efforts, the industry is still in crisis. As software complexity increases, the core problems are exacerbated.

This article focuses on the question of "efficiency", from the perspective of three criteria: time, cost and quality.

1.1 Analysis of the Software Development Process

The typical software development process can be represented conceptually as a multi-level human communication channel.

Illustration 1: Multi-level Human Communication Channel



The customer communicates with the BA, the BA with the designer, the designer with the developer, and the developer with the tester.

At each communication level, information is analyzed by an assigned specialist and transformed into a new format: at the BA level, into the requirements document, at the Design level, into the design specifications, at the Developer level, into the code and, at the Tester level, into the tests.

As we can see from the model, performance and quality of the entire process is dependent on:

- performance and quality parameters at each level of information processing; and
- performance and quality parameters of communication between levels.

Thus, efficiency of the software development process depends on:

- the efficiency of each group of software development specialists; and
- the efficiency of human communication between the groups of specialists.

1.2 Analysis of Working Efficiency of Software Specialists

Recently, there has been an enormous industry-wide effort to improve quality in the work of software specialists.

1.2.1 The Empowered Developers – the "First Born"

Particular attention has been paid to improving the work of developers:

- more efficient methodologies and standards are being proposed and published;
- new software development tools are appearing on the market; and
- many professional seminars and workshops are being developed and offered.

Since programming was really the first computer profession, today's developers benefit from being the "first born" among other software specialists, enjoying the overwhelming bulk of attention and improvement efforts.

As a result, developers are fully equipped to perform their work efficiently. In general, "**the empowered developers"** already have sufficient tools, methods, models and knowledge.

1.2.2 The Semi-Equipped Tester

Y2K played a significant role in establishing testing as a self-contained discipline. More efficient formal testing methods and techniques are now replacing ad hoc approaches and creating new opportunities for test improvement. However, testing and QA improvement is still hampered by serious limitations in modern testing tools. While test tools can automate test execution, there are virtually no tools on the market for automating the most difficult and intellectually demanding part of the testing process: *test development*.

We will define this issue as the **"semi-equipped tester"** (a tester equipped only with limited tools and resources).

1.2.3 The Unequipped Business Analyst

Business analysts are in a much more difficult situation than developers and testers. They still suffer from a lack of efficient standard methods of requirements development, analysis, integration and change. The introduction of the Use Case, a relatively new and very popular requirement development artifact, is a step in the right direction. But in practicality, the typical business analyst has great difficulty developing Use Case models, due to the absence of Use Case definition standards and methods for Use Case development and validation. As a result, business analysts typically use ad hoc approaches. The results of their work remain largely dependent on subjective factors, including individual skill levels and other personal characteristics.

We will use the term **"unequipped business analyst"** to reflect this situation.

1.2.4 The Powerless Customer

Of all roles in the software development process, customers are unquestionably in the worst situation. There are still no accepted methods or tools to help the customer define, refine, analyze, validate and express their needs. In the software development business, customers are forced to articulate what they want in exactly the same antiquated way as they have done in other contexts for hundreds of years. We will use the term "**powerless customer"** to reflect this situation.

An analysis of working methods and techniques used by different specialists involved in the various phases of the software development life cycle shows not only that the efficiency of their work varies, but also that *the customer and the business analyst have become the most critical elements of the entire process*. Ironically, it is the customer and business analyst who have been largely ignored in recent quality improvement initiatives. Few, if any, new tools or methodologies have been introduced to assist them in their stage of the process.

1.3 Ambiguity in Human Communication

Even if we see improvements in the working efficiency of software specialists, a fundamental problem persists: fallible human communication. Today we can organize very reliable communication between rockets and submarines, airplanes and automobiles; between different types of computers and telephones. We have wrapped the world in multiple communication lines, we have Intranets and the Internet. But we still tend to misunderstand each other, even when speaking the same language. We will call this phenomenon **"ambiguity in human communication"**.

Ambiguity in human communication is often the root cause of low performance and poor quality of the software produced by the contemporary software development process. Even if all other reasons for software development efficiency problems are eliminated, errors arising from an imperfect understanding of customer needs will continue to be the major constraint on software development improvements.

We demonstrate below that no design/code/testing effort can meaningfully improve the quality of software products until these roadblocks are removed.

1.4 Core Reasons for Low Efficiency of Software Development

As illustrated by the analysis in Sections 1.1 – 1.3, the core reasons for today's software development inefficiencies are:

- ambiguity of human communication;
- the powerless customer;
- the unequipped business analyst; and
- the semi-equipped tester.

These reasons are listed in order of importance. However, ambiguity of human communication is the over-arching and pervasive issue in all phases of the software development life cycle.

When one factors in the typical cycles of change, integration and evolution associated with the development of any software product, the problems caused by these core reasons for low efficiency are significantly magnified.

These problems translate into huge costs to today's businesses. According to a recent study by the National Institute of Standards and Technology (NIST), a federal agency within the U.S. Commerce Dept., "Software bugs are costing the US economy an estimated **$59.5 billion** a year". And there is more than just the cost of low quality to consider; there are also the dire economic consequences of the software development industry's continuing failure to deliver solutions on time.

Below, we demonstrate that even the most aggressive design/code/testing improvement efforts will not be enough to increase the efficiency of software development until the root causes of its current inefficiency are eliminated.

1.5 Paradoxes of the Software Development Process

1.5.1 Requirements have become a prototype of future software problems

As a result of the "ambiguity of human communication", the "powerless customer" and the "unequipped business analyst", articulated software requirements typically fail to reflect the customer's needs (see Illustration 1).

Incorrect requirements lead to defective software.

Problems in software arising from incorrect requirements cannot be identified during the testing processes, because tests developed from incorrect requirements and will inevitably have exactly the same bugs.

As a result, instead of being a prototype of software that meets customers' needs, requirements all too often become a prototype of future software problems!

1.5.2 Tests, instead of finding bugs in the product, are harboring bugs themselves

There are three reasons why the benefits of testing can quickly become illusory:

- First, when tests are developed based on requirements, they inherit the same problems and limitations as the requirements and the implemented product. Applying these incorrect tests to the product will result in the misleading conclusion that there are no bugs.

- Furthermore, even with absolutely perfect requirements, tests still will be incorrect as result of the problem we have identified as the “semi-equipped tester”. For today’s complex systems, manual test development is simply not adequate. Manual test development will not result in accurate, comprehensive testing of software.

- Finally, if tests are developed based on the ultimate product, rather than the requirements that it was supposed to satisfy, tests would still have these same errors and omissions. The product will be defective because it will be plagued with the same errors exhibited by the requirements. Product-based tests will not expose these defects (see paradox 1.5.1).

Applying tests to the product in this context will never identify all the product’s bugs, omissions and problems. Tests, instead of finding bugs in the product, are harboring bugs themselves.

1.5.3 Design/coding process improvements do not allow development of a bug-free product

Even if one were to assume a perfect design/code process – completely free of bugs – the resulting product still has bugs. Why? The requirements, which are the input for the design/coding process, are incorrect (see paradox 1.5.1). Incorrect inputs lead to deficient outcomes.

1.5.4 Each modification produces an exponential number of other modifications

The introduction of modifications to requirements raises the same problems as baseline requirements development: “ambiguity of human communication”, “the powerless customer”, and “the unequipped business analyst”. As a result of the modifications being incomplete and incorrect, further modifications are required.

Modifications introduced in the design/code phase of the software development life cycle result in inconsistencies between requirements and the implemented product, increasing the need for subsequent modifications and making the modification process itself more difficult, less controlled and incapable of proper management.

1.5.5 New development is really legacy systems development

The traditional software development process all too often produces incorrect requirements, a faulty product and inconsistent documentation. Really we are creating out-of-control systems that become “legacy systems” even before they are delivered!

2. Modern Methods for Improving Software Development Processes

The following are two of the most popular current attempts to improve the software development process:

- CMMI (Capability Maturity Model Integration); and
- Agile Development.

It is important to understand how effective these methods are at attacking the core reasons for the low efficiency of software development, as formulated in Section 1 of this article.

2.1 CMMI

Capability Maturity Model Integration (CMMI), the highly respected software management methodology published by the Software Engineering Institute (SEI), has been adopted by thousands of major organizations seeking to position themselves as industry leaders in software development.

CMMI has been shown to reduce risks associated with development projects, increase efficiency and improve the quality of software deliverables.

CMMI introduces a set of goals and practices geared to improving the software development process. It has been developed for the classical software process, which can be represented by the multi-level human communication channel (see Illustration.1).

2.1.1. CMMI, the "The Powerless Customer" and "The Unequipped Business Analyst"

Solving the problems of "the powerless customer" and "the unequipped business analyst" is the principal objective of CMMI Goals in the following CMMI Process Areas:

- Requirements Management; and
- Requirements Development.

Table 1, below, defines these CMMI Goals.

Table 1: CMII Goals and Practices, in Requirements Management and Requirements Development

Process Area	**Maturity Level**	**Goals**	**Practices** (In each SP X.Y-Z, the Z specifies Capability level.)
Requirements Management	2	SG 1 Manage Requirements	SP 1.1-1 Obtain an Understanding of Requirements
			SP 1.2-2 Obtain Commitment to Requirements
			SP 1.3-1 Manage Requirements Changes
			SP 1.4-2 Maintain Bi-directional Traceability of Requirements
			SP 1.5-1 Identify Inconsistencies between Project Work and Requirements
Requirements Development	3	SG 1 Develop Customer Requirements	SP 1.1-1 Collect Stakeholder Needs
			SP 1.1-2 Elicit Needs
			SP 1.2-1 Develop the Customer Requirements
		SG 2 Develop Product Requirements	SP 2.1-1 Establish Product and Product-Component Requirements
			SP 2.2-1 Allocate Product-Component Requirements
			SP 2.3-1 Identify Interface Requirements
		SG 3 Analyze and Validate Requirements	SP 3.1-1 Establish Operational Concepts and Scenarios
			SP 3.2-1 Establish a Definition of Required Functionality
			SP 3.3-1 Analyze Requirements
			SP 3.4-3 Analyze Requirements to Achieve Balance
			SP 3.5-1 Validate Requirements

The recommended practices shown in Table 1 are useful guidelines for improving the process of capturing requirements. But several very important practical questions remain: How does one implement and follow these practices? How does one apply these guidelines to “obtain an understanding of requirements”, “analyze requirements”, or “validate requirements” without modern models and tools to help us?

To be more precise, how should one implement specific practices such as “develop the customer requirements” and “establish product and product component requirements”, which call for bi-directional traceability between requirements? The achievement of bi-directional traceability, when attempted manually, is a very difficult, time-consuming and error-prone process. Even more difficult is the challenge of modifying traceability when requirements change.

Thus, although CMMI has a disciplined engineering approach to requirements gathering and requirements development, it is also an approach that cannot be implemented efficiently or reliably without modern analysis models and tools. This statement was confirmed by many participants of SEPG 2004-Enterprise Process Improvement Conference, with whom the author had opportunity to discuss these concepts.

As a result, poorly implemented CMMI (without supporting models and tools) does not allow us to overcome the problems of “the powerless customer” or “the unequipped business analyst.”

2.1.2 CMMI and Human Communication Ambiguity

Conceptually, CMMI employs an age-old principle - “Divide and Conquer” - to rationalize the software development process. All processes are subdivided and deconstructed into process areas; process areas are broken down into goals; goals into practices; practices into steps. The intent is to introduce more discipline to the entire software development process, including human communication, and to improve its quality.

Some degree of human communication improvement can be achieved with CMMI’s verification and validation practices. But the same limitations identified in 2.1.1. (i.e. an absence of supporting models and tools) prevent us from attaining a complete solution for the human communication problem.

2.1.3 CMMI and “The Semi-equipped Tester”

CMMI does not provide any specific methods or tools for automating the test development process. As a result, it does not solve the problem of “the semi-equipped tester”.

2.1.4 Summary of CMMI Analysis

The results of an analysis of CMMI's ability to attack the core reasons for software problems are represented in Table 2. The table considers each core reason and CMMI's ability to eliminate it, classified into 3 levels of capability:

Level 0 – not capable of eliminating this core reason
Level 1 – partially capable of eliminating this core reason
Level 2 – capable of eliminating this core reason

Table 2

Core Reason for Software Problems	**Capability of Elimination**	**Limitation**
The Powerless Customer	0	Absence of supporting models and tools
The Unequipped Business Analyst	0	Absence of supporting models and tools
The Semi-equipped Tester	0	Absence of supporting models and tools
Ambiguity of Human Communication	1	Absence of supporting models and tools

As the analysis above shows, CMMI does not resolve the core problems inherent in software development. CMMI's software process improvement capabilities are restricted by the absence of supporting models and tools. This is why practical implementation of CMMI has proven to be very expensive, and why many organizations have spent years attempting to advance even one CMMI level.

2.2. Agile Development

Agile development methods (such as XP, Scrum, Crystal Orange, DSDM, Adaptive Software Development, Feature-Driven Development, and "pragmatic programming") have been proposed as a response to the CMMI problems of "postponed" delivery, communication, requirements, testing and documentation.

Agile methods are defined in the Agile Manifesto [1] as:

"Individuals and interactions over processes and tools;
Working software over comprehensive documentation;
Customer collaboration over contract negotiation;
Responding to change over following a plan."

To analyze how Agile methods attack the core reasons for software development problems, we will look at the example of Extreme Programming (XP).

In the XP methodology, the customer is fully involved in the software development process. He works with programmers in the same room throughout the entire project. The customer describes what he needs in small simple increments called "user stories", which are about 3 sentences of text. Implementation of user stories is performed iteratively.

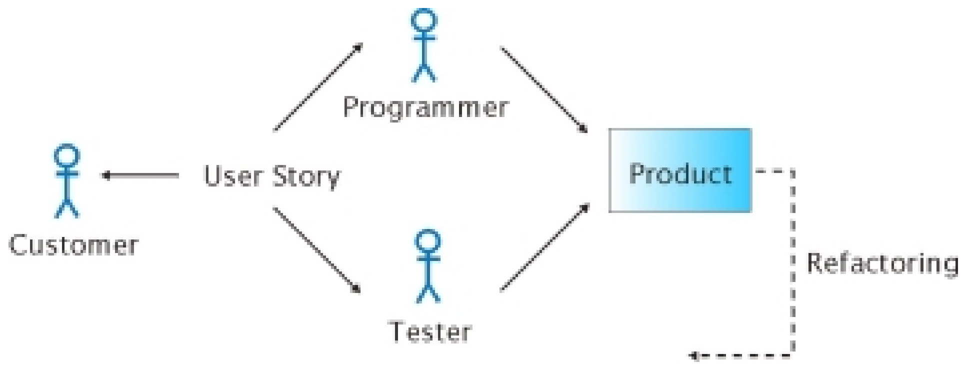
Developers work in pairs, and implement these stories into code. One story results in one iteration. Iterations tend to be of short duration; generally 1 to 3 weeks.

Simultaneously, testers write tests for each story and are ready to test it even before the code is written, following the principle "test before code".

All processes are followed by refactoring, which involves "altering the structure of an existing code base to improve its design quality" (Martin Fowler, [2])

One can adapt the multi-level human communication model to the XP method for analysis. A simplified model representation of one iteration of the XP process is represented in Illustration 2.

Illustration 2: Single iteration in XP process



As one can see from the picture, this model varies significantly from the CMMI model (Illustration 1). The relevant question, however, is whether or not this change in approach really improves the software development process, eliminating the core reasons for the problems identified above.

2.2.1 XP and the "The Powerless Customer"

XP proposes that the customer be a part of the development team and spend all his time on the project. The customer sits with the development team in the same office and has the opportunity to discuss his needs with programmers and testers directly. The customer is also directly involved in "User Acceptance Testing".

This sounds like a movement in the right direction. But does XP offer the customer some methods or tools to understand his own needs, and to express them in the clearest manner? Does XP really ensure that there is some opportunity to validate, from the beginning of the process, that the customer is being understood properly?

The answer to both these questions is "No". To sit in the same room is one thing, but to understand each other is something else entirely. Language is more important than location in this sense. Consider the problem in this way… is communication better when one speaks to the person who one sits with every day if that person does not really speak or understand your language, or is it better to communicate in a common language with someone even at a far distance? It seems that XP, instead of equipping the customer with new methods and tools to improve his effectiveness and efficiency, takes up a large amount of his time with no guarantee of meaningfully better results.

However, XP does present a very important new opportunity for the customer. The customer can now see prototypes of parts of his system every 1-3 weeks (i.e. after each new iteration). This allows the customer to validate his ideas regarding each new piece of functionality, and to confirm that his ideas have been correctly interpreted and understood by programmers. Even though this validation can be performed only for pieces of system functionality, and only after their implementation, this is still a real step toward elimination of the "the powerless customer" syndrome.

2.2.2 XP and the "The Unequipped Business Analyst"

XP does not assume the use of a business analyst (BA) in the development process. As one can see from Illustration 2, there are fewer process roles in XP, compared to CMMI.

"But if there is no BA, who develops requirements?" one might ask. The answer is… no one. XP promotes the idea of minimal documentation, and focuses on working code instead of documentation. This raises two fundamental issues:

1. XP was heralded as a response to change management problems within CMMI and similar software development approaches. But how does one maintain XP-developed products and introduce changes to them *without* requirements documentation?

2. Everyone knows the nightmare of working with legacy systems, due to the typical absence of documentation. Since XP places little emphasis on documentation, are XP promoters introducing classic and intractable "legacy problems" into new development?

Some XP promoters offer the possibility of using a BA in the customer role, but this is less efficient and potentially less reliable than using the actual customer. "Although the business analysts have a good understanding of the business, they may lack the detailed, leading-edge knowledge that would let them steer the project to a truly superior solution."[3]

Thus, XP not only fails to solve the problem of "the unequipped business analyst" problem, it exacerbates the problem by replacing deficient requirements with a complete absence of requirements.

2.2.3 XP and the "Semi-equipped Tester"

XP pays serious attention to testing. "Tests are created before the code is written, while the code is written, and after the code is written". We will consider the impact of XP methods on the problem of the "semi-equipped tester", for individual iterations and for the product releases.

Individual Iterations: XP improves test quality for individual iterations, by decreasing the complexity of the test development task. Test development is based on the short, simple user stories. Testers also have the opportunity of direct contact with customers, and to revisit and improve developed tests.

Product Releases and Entire Product: It is easy to demonstrate that XP methods do not allow development of comprehensive tests for the product releases or for the entire product. Products (product releases) are generally created as a result of multiple development iterations. Refactoring follows each new iteration and supports continuous code improvement, although not necessarily ensuring that the code correctly performs the required functionality.

The situation with tests is different. Assume that tests exist for a first and second iteration. What about *integrated* tests for the two iterations once integrated together? What should one use as source information for the tests? Requirements? There are no requirements. User Stories? There are no integrated user stories. One can only create tests based on assumptions or by using ad hoc methods. However, the quality of these tests and their results, is questionable. And of course, this problem compounds itself upon an increasing number of iterations.

Thus, XP only partially eliminates the problem of "the semi-equipped tester".

2.2.4. XP and "Ambiguity of Human Communication"

XP methods improve human communication by introducing more intensive human interaction in the software development process. But an absence of efficient methods and tools that would support a better understanding between groups of specialists, prevents the "human communications ambiguity" problem from being solved completely.

2.2.5. Summary of Agile Development Analysis

The ability of Agile development methods to attack the core reasons for contemporary software problems, using the example of XP, is represented in Table 3.

Table 3

Core Reason for Software Problems	Capability of Elimination	Limitation
The Powerless Customer	1	Absence of supporting models and tools
The Unequipped Business Analyst	0	BA is eliminated from process
The Semi -equipped Tester	1	Absence of system requirements
Ambiguity of Human Communication	1	Absence of supporting models and tools

Agile methods propose some meaningful improvements but do not completely eliminate the core reasons for today's software problems.

It is also important to note that Agile methods introduce a new quality problem as result of their "non holistic" approach to software development. For example, refactoring, which is intended to improve quality of the code, very often has negative implications for quality. Refactoring expert Martin Fowler explains the role of refactoring in XP programming in this way:

"With Extreme Programming, you use refactoring continuously, so every day while doing some refactoring to the code you're writing so that whenever you start a task, you look at the code and say, does the code work in a way that allows me to add this new piece of functionality I need? If not, you refactor it to make it easier. Also, while you're getting the test to work, you're not quite so concerned about the design quality; you're concerned about getting the function in. But then as soon as you've gotten the test to work, you must refactor to make the design quality very, very high." [2]

But where is the guarantee that during this "past test" refactoring, one will not introduce new bugs? As demonstrated in 3.2.3 above, one cannot efficiently find system mistakes using XP tests. These mistakes can be accumulated such that the probability of system correctness will exponentially decrease with a growing number of iterations. This is one of the reasons why Agile methods can be applied only to relatively small projects.

Thus, our analysis of CMMI and Agile methods concludes that that they introduce the possibility of some improvements without resolving the core reasons for contemporary software development problems.

These methods introduce improvements based mostly on organizational changes and disciplined activities, but what the software industry really needs are ***fundamental technical changes*** and ***new execution paradigms*** for the successful development of complex modern software systems.

3. Customer Oriented Software Development (COSD)

Customer Oriented Software Development (COSD) describes a new software development paradigm which fundamentally changes the entire software development process, altogether eliminating the core software problems discussed in this paper.

What makes COSD fundamentally different? COSD shifts the focus to supporting and improving the process of “ideation” (the forming of ideas) and the transformation of idea to reality.

3.1 Theoretical and Practical Foundation

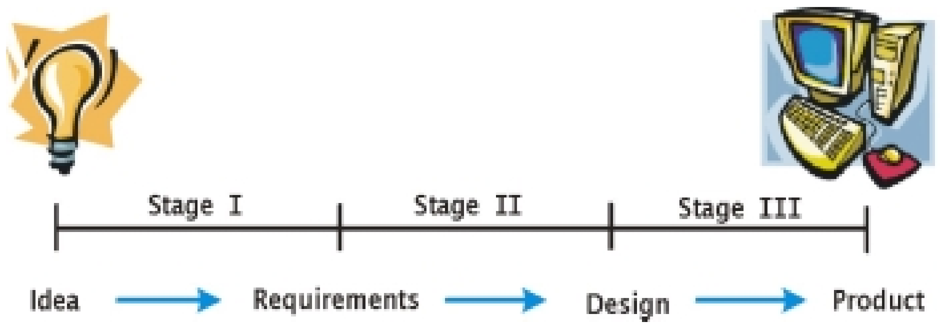
One should first consider the software development process philosophically. What is it?

Fundamentally, it is a process of transforming elements of the ideal world into elements of the material world, with human participation. It is the transformation of ideas into a finished product.

Today, this “Idea to Product Transformation Process” consists of 3 main stages:

- Stage I: Idea $\rightarrow$ Requirements (Ideation Stage);
- Stage II: Requirements $\rightarrow$ Design (Design Stage); and
- Stage III: Design $\rightarrow$ Final Product (Coding Stage).

Illustration 3: Stages in Idea to Product Transformation



It is essential to analyze these stages in some depth.

3.1.1. Stage 1 Idea → Requirements

This stage is the most important, because at this stage the first embodiment of the idea emerges in the material world.

3.1.1.1 Business Analysis and Requirements

Originally, in the beginning of the programming era, programmers themselves could understand, create and implement ideas. This was possible because the first programming tasks were usually expressed mathematically and therefore had formal representation.

Later, as result of the continued development of our technological civilization, more and more complex ideas and idea systems needed to be transformed into software. A growing problem for software development inevitably emerged: how can one communicate complex ideas properly and what sort of specialist can understand and translate such communication?

For example, if a customer needs to create software for a complex business system, how does one effectively explain to programmers the essential business needs, concepts and processes being addressed? The conceptual worlds of business people and programmers are very different. One side is a “world of ROI” and the other side is a “world of statements and loops”. The communication gulf is understandably enormous.

This “disconnect” between the conceptual worlds of business and programmers necessitated the introduction of a new intermediate specialist – the Business Analyst (BA). The conceptual world of the BA includes some elements of conceptual worlds of both the customer and the programmer. The major task of the BA is to understand customer ideas and to reflect them as a set of documented “requirements” which, in turn, should be understandable by the programmer. Thus the BA performs the transformation or translation of ideas from the conceptual to the material world.

Theoretically, the requirements are the first embodiment of the “idea” in the material world.

Practically, the requirements are a common resource for information for the entire development team, as shown in Illustration 4.

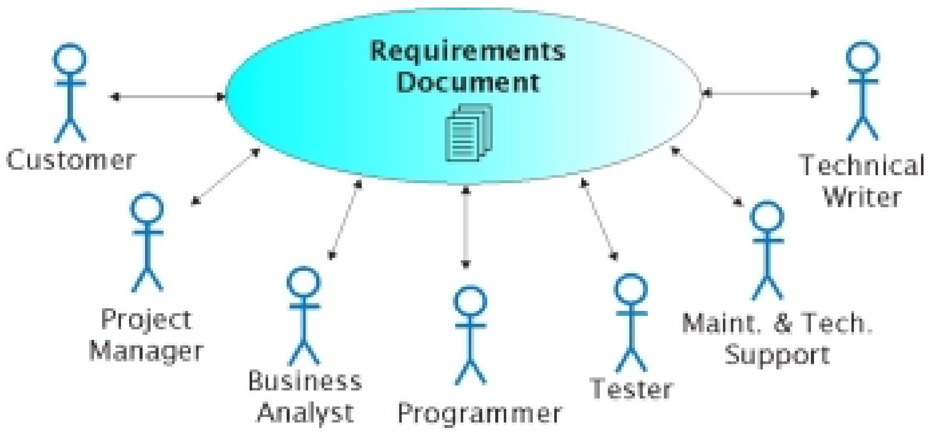


Illustration 4: The requirements document is a common source of information for the entire development team

Below, Table 3 shows how the team members use requirements for their work.

Table 3 - How requirements are used by team members

A person in this role...	Uses the Requirements document...
Customer	to verify that his needs were understood properly, and to validate the scope and volume of work
Project Manager	to assign tasks and to schedule and manage their completion
Programmer	as a major source of information for the development of the program specifications and code
Tester	to create test scenarios and test cases for the product validation
Maintenance & Technical support	to analyze problems and introduce change
Technical Writer	as a source of information for User Manuals and other documents

As quickly becomes apparent, the correctness and completeness of project requirements determines the quality of all software artifacts. But the quality of requirements is typically very questionable.

3.1.1.2 Requirements Quality

What criteria allow requirements to qualify as good requirements?

Within the scope of any “idea-to-requirements transformation process”, requirements must first meet the criteria of *accurately reflecting customer needs*. But what is the situation in today’s reality?

- The customer does not often clearly understand his own needs. He has some overall vision of his future system, and some partial pictures of its behavior.
- The customer’s explanation will likely have elements of redundancy, incompleteness and misconception.
- The BA frequently introduces errors due to misinterpretation of the customer’s explanation.
- The BA also introduces his own bugs in requirement specification, as typical human error.
- After many meetings and discussions the requirements are “approved” or “signed off”, and everyone assumes they have good requirements specifications, which is not the case.
- The customer cannot properly or easily validate the quality of the resulting requirements, nor validate that the requirements match his original ideas.

As a result of these factors, the quality of the requirements is extremely low.

3.1.1.3 Idea → Requirements Transformation Risk

The “idea to requirements transformation process” is the most difficult and unreliable part of the entire software development life cycle.

Typically, this transformation is performed manually and the resulting requirements documents contain misinterpretations, errors and omissions.

The most difficult questions are pervasive:

What is the customer idea?
How can one correctly ascertain and understand the customer idea?
How can one articulate a correctly understood customer idea?

Until these questions are answered, it is practically impossible to develop correct requirements for any software development project.

As shown below, COSD delivers a direct and clear answer to these critical questions.

3.1.2 Stage 2 Requirements → Design

In this section, we consider the transformation of requirements not only into the design specification, but also into test specifications (even though testing is typically considered as a separate stage in the software development lifecycle). We take this approach because both of these transformations can and should be performed in parallel.

3.1.2.1 Requirements → Design Transformation

Programmers use design specifications as input artifacts for their coding. Today's design is typically represented by flow charts, structural diagrams for the procedural approaches, activity diagrams and other UML artifacts for object-oriented approaches.

Because the typical requirements document is not very formal, there is vast opportunity for misinterpretation by developers. The manual development of program specifications creates a high risk of introducing further misinterpretation errors. When requirements are incorrect or incomplete, the likelihood that additional errors will be introduced during the design phase is even higher, as a result of the ambiguity of human communication. If a programmer sees some contradictions or omissions in the requirements, he tries to resolve the problem through multiple discussions with the BA or even the customer. Very often, even when the right answer is found, it is not reflected in updated requirements, and different specialists end up using different versions of requirements.

3.1.2.2 Requirements $\rightarrow$ Test Specifications Transformation

A similar situation emerges in the transformation of requirements into testing specifications. But in this case, in addition to the possible errors mentioned above, there are many errors caused by incorrect test specification development processes.

Transformation of requirements into test specifications is technically much more complex than transformation or requirements into program specifications. In fact, programming specifications are received as result of the natural transformation of requirements. In the case of test specifications, the base challenge of compilation is compounded by the very complex process of synthesizing tests and test procedures.

3.1.2.3 Requirements $\rightarrow$ Design Transformation Risk

As sections 3.1.2.1 – 3.1.2.2 demonstrate, the transformation of the requirements to the design (including, especially, test specifications) is a risky and unreliable process.

These risks are due not only to the poor quality of the requirements themselves, but also due to mistakes related to manual interpretation of the requirements.

In replacing manual methods for developing requirements, by introducing design transformation with automated methods, one is able to significantly reduce the risks of misinterpretation by developers.

Correspondingly, the introduction of automated test development methods will dramatically reduce the risks related to the transformation of requirements into tests.

As it will be shown in Section 4, COSD proposes automated methods for generating design and test artifacts.

3.1.3 Design $\rightarrow$ Final Product Transformation

The final stage of the software development process is the transformation of the design specification into the actual product – the coding process. Correspondingly, test specifications are transformed to real test cases that can be applied to product testing.

3.1.4 Overall Risk Analysis

As we have seen, the Idea to Product transformation process consists of 3 stages:

- Stage I: Idea → Requirements;
- Stage II: Requirements → Design; and
- Stage III: Design → Final Product.

Usually, the success of the overall transformation from idea to final product is validated by testing of the final product. Test specifications are prepared after, or simultaneously with, the development of the program specifications.

Where do the greatest risks of failure arise?

- The first stage, the "idea to requirements transformation process" or "ideation phase", is the most risky stage of the whole software development process. At this stage, we make the biggest leap from the "ideal" or "conceptual" world to the "material" or "actual" world, and we have the largest number of transformations (i.e. translations):

  Customer idea → customer words → BA words → BA ideas → BA requirements

- The second stage, the "requirements to program and test specifications transformation process" or "design phase", is risky too, but the risk here is decreased by the fact that transformation is performed entirely within the boundaries of the "material" world.

- The third stage of the process, the "program specifications to final product transformation process" or "coding phase", is the least risky, because it involves transformation from one formal representation to another, such as from flow chart to code.

Historically, experts have paid most attention to the third stage of the software development life cycle, which actually presents the lowest risk. Given the overtly programmer focused approach traditionally taken in the development of software development methodologies and tools, it is not so surprising that the industry still lacks effective methodologies and tools for increasing reliability of the most risky first two stages of the "idea to product" transformation process.

However, despite the fact that some experts have recently begun to turn their attention to the problem of requirements definition and, in particular, to reducing the errors resulting from that stage of the software development life cycle, their focus still tends to be on the second stage – the "design stage", when the *real* source of problems is even earlier in the software development life cycle.

To resolve the current crisis in software development, we need to address the root cause of low efficiency by ensuring that the customer's idea is fully understood and comprehensively articulated well before the design or coding phases of the process begin. This is the only future paradigm for software development that can adequately address the massive "transformation risks" inherent in the software development life cycle.

3.2 Idea of the Future: COSD Paradigm

In this section, a fundamentally new approach to software development – Customer Oriented Software Development (COSD) – is introduced and explained. COSD, a methodological reworking of the traditional software development process, was created to solve cost/time/quality problems in the software development life cycle by eliminating the core reasons for them.

Because the COSD paradigm assumes that the customer and his ideas are of prime importance, implementation of COSD allows us to:

- Help the customer understand his own ideas;
- Help the customer to evolve/refine his ideas and to more clearly formulate his needs;
- Help the customer explain/articulate his ideas so that they are understood by other team members;
- Give the customer the opportunity to validate/confirm that he was understood properly;
- Give the customer appropriate project visibility and control;
- Allow the customer to see the impact of changing/modifying his idea and to compare different ideas in order to choose the best solution; and
- Fully involve the customer in the project, without monopolizing his time.

Coming to a clear understanding of the customer "idea" is only part of the challenge. As managers of complex software development projects all know, an initial product "idea" can easily be distorted or lost by the subsequent stages of the software development process. COSD allows software development organizations to avoid such results by ensuring that project artifacts do not breach the scope of the original idea unless the original idea has changed.

COSD facilitates automated real-time parallel development of project artifacts, including documentation and tests, based on the evolving and iteratively validated customer idea.

And because the customer can (and often does) change his original idea, COSD deals with change as a necessary attribute of the modern software development process. COSD ensures organic, comprehensive and coherent change of all project artifacts when the original customer idea changes at any stage in the product development process.

.

3.2.1 What is COSD?

COSD is the method for discovering, evolving, refining, validating and managing the continuous transformation of the customer idea into an executable software product, based on the Universal Diagnostic Imitation Model.

COSD assumes:

- The possibility of documentation and validation of the transformation process, at any point in the software development life-cycle, against the original customer idea;

- The introduction into the software development process of a stage of enquiry and analysis (called the "customer needs determination" phase) that is antecedent and prior to the requirements determination and articulation phase. The process of "customer needs determination" enables the ideation of the customer's perception of the future product in his own language and terminology. The result of this process automatically serves as a requirements prototype.

- A new type of software development process, the "parallel process", which replaces the serial process typical of CMMI methods and the refactoring process essential to Agile methods. The "parallel process" allows concurrent and automatic generation of all software development project artifacts, including requirements, design specifications, product tests and the product itself, based on customer needs.

- Overall bi-directional traceability between all project artifacts and customer needs;

- Coherent change of all project artifacts in tandem with any modifications to or enhancement of customer needs; and

- Integration, at a high level of abstraction, of ideas, needs, and requirements.

Basic concepts of the COSD paradigm are represented by illustrations 5 and 6.

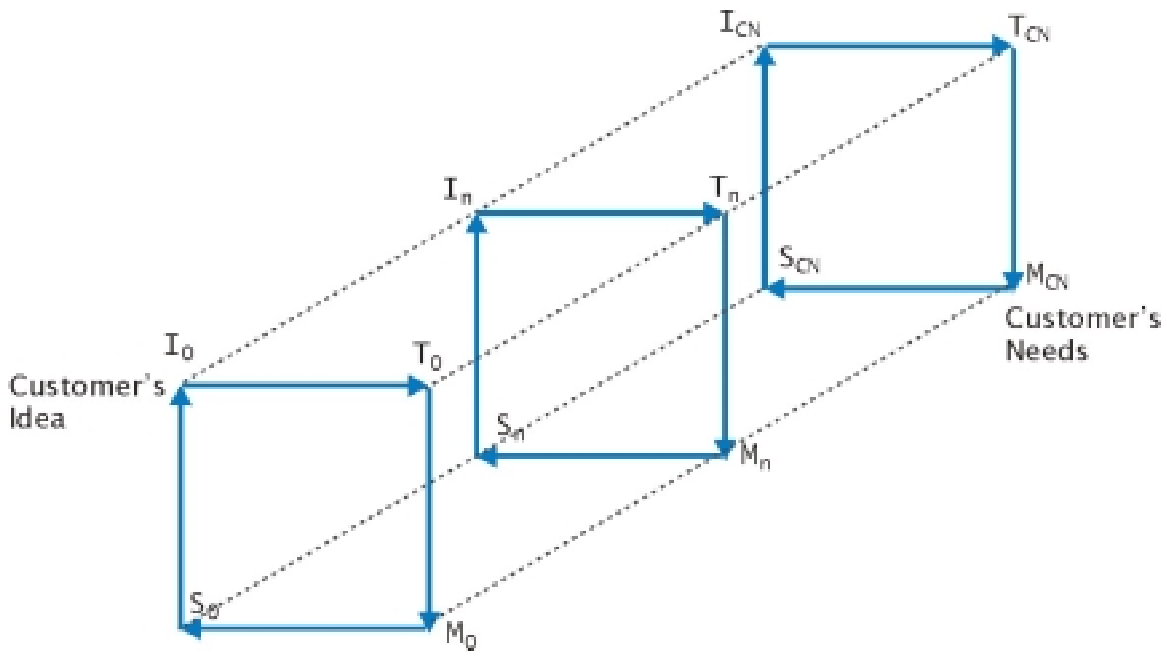


Illustration 5 - Evolution, validation and transformation of Customer Idea into Customer Needs

The original customer Idea, I_0, is translated by T_0 using intermediate language into the Universal Diagnostic Model (M_0). Model M_0 allows verification of the original customer idea, and supports automated retranslation of it into dynamic simulation view (S_0), which is represented in the customer's language / terminology. The customer reviews the simulation and validates it against his original idea (I_0). As a result of this validation process, two outcomes are possible:

- The customer does not agree with the representation, and concludes that his idea was mistranslated; or

- The customer sees new possibilities for his future product, and evolves his original idea.

In the first case, sequence I_0, T_0, M_0 and S_0 will be repeated and corrected.

In the second case, original idea I_0 will be transformed into the idea I_1, and the entire validation process will be repeated via T_1, M_1, S_1 states. It is of utmost importance in this process that states I_n, T_n, M_n and S_n are fully consistent and compliant with each other. This idea validation and evolution process will be repeated as many times as needed, until the customer is satisfied with his idea of the product. This significant moment is defined by I_{CN}, T_{CN}, M_{CN} and S_{CN} states, where "CN" signifies "customer needs" (Illustration 6).

At this stage the Model fully reflects customer needs and can be used as the information generation engine for all project artifacts and for the software product itself.

Illustration 6 shows how COSD changes the software development life cycle.

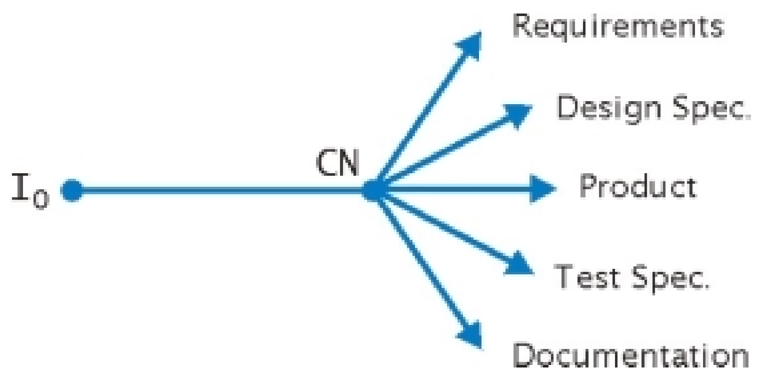


Illustration 6: Parallel Software Development Lifecycle

The process consists of two basic stages:

- evolving the original customer idea to the state of customer needs; and
- concurrent generation of all product artifacts based on customer needs, including the product itself.

Thus, instead of the traditional approach in which we manually develop product artifacts based on incorrect customer requirements that rarely reflect customer needs, COSD proposes the possibility of automated and concurrent generation of all artifacts based directly on validated customer needs.

As a result of this approach, the seminal importance of the requirements phase of the software development life cycle is diminished. In effect, requirements become a documentation source only (i.e. another artifact flowing from customer needs), useful for future product maintenance and support.

It is easy to see that, using the COSD approach, the time required for any complex development process is significantly reduced when compared to traditional processes. At the same time, the COSD methodology guarantees a very high quality of final product, because the transformation from idea to final product is not only automated; it flows directly from validated customer needs.

As a result, the “centre of gravity” in the software development process shifts from coding (i.e. the programmer) to the customer, conferring total control to the customer over the entire software development process.

COSD eliminates all of the “core reasons” for contemporary software development problems:

- powerless customer;
- unequipped business analyst;
- semi-equipped tester; and
- ambiguity in human communication.

As a direct result, COSD also significantly increases the overall efficiency of the software development process. In particular:

- The customer receives tools to evolve, refine, validate and express his ideas;
- The business analyst can automatically generate requirements based on customer needs;
- The tester has automatic test generation capability; and
- Ambiguity in human communication is eliminated due to the automation of the transformation from customer idea to final product.

Table 4 – COSD Capability to eliminate the core reasons for software problems

Core Reason for Software Problems	**Capability of Elimination**	**Limitation**
The Powerless Customer	2	None
The Unequipped Business Analyst	2	None
The Semi-equipped Tester	2	None
Ambiguity of Human Communication	2	None

Profesy™ is the first in a series of tools from Sofea Inc. that automates COSD methodology. Profesy™ performs automated requirements generation, simulation and validation, as well as automatic generation of Flowcharts, Activity Diagrams, Use Cases, Tests, and Documentation.

Profesy™ also supports all contemporary methods, such as RUP, CMMI, Agile Development, MDA, SOA and SODA.

Customer Oriented Software Development, powered by Profesy™, will change the way that all complex software is built and modified. Profesy’s ability to accurately capture and simulate the customer’s idea of a future software product is, itself, an idea about the future of software development whose time has come.

References:

[1] "Agile Manifesto"
www.agilemanifesto.org

[2] Fowler, Martin. "XP (Extreme Programming)"
www.martinfowler.com

[3] Hayes, Steve. "An Introduction to Agile Methods"
Steve Hayes (Khatovar Technology)
steve@khatovartech.com
http://www.khatovartech.com

Additional Resources:

Passova, Dr. Sofia. "CMMI and Profesy: How Profesy Supports CMMI Goals and Practices in Engineering Process Areas"
www.sofeainc.com

Orr, Ken. "CMM versus Agile Development: Religious Wars and Software Development", Cutter Consortium.
www.cutter.com/freestuff/apmreport.html

Andrews, Martin (Object Consulting).
www.objectconsulting.com.au

Jeffries, Ron. What Is Extreme Programming? 2001
www.xprogramming.com