Software Inception with TRIZ

Edgar Daniel Fernández Rodríguez

Investigador Líder: Cuauhtémoc Lemus Olalde

Table of Contents

Table of Figures

1. Introduction

Software nowadays faces a major challenge: to satisfy the growing needs of users. That is: add more and more functionality into a single application or the interaction of many applications to offer the most complete set of functions to users. Software industry is not concerned anymore for Processing Power and Resource Use efficiency, but to solve user needs without making complex solutions.

For this, Software needs a Systematic Method for Problem Solving and a way to find new solutions for the new problems and challenges Software Engineers are facing.

TRIZ is a method that has been used for many years to develop new products and patents, and has proved to be very effective in the Physics, Chemistry, Mechanics and Medicine Fields. The fundamental principle, as stated by Altshuller in **[1]**, is to find a particular solution from a general one, even though it was applied in a different field.

Architecture Design is one of the phases of the Software Development Lifecycle where most of the solution is defined. Some Quality Models and Process Definitions include the need for Architecture and Evaluation need, provide some guidelines of what needs to be done, but none defines how to do it.

Architects face the problem that they know what must be done, but lacks tools and methods to do it systematically. We'll define some of the problems and causes on section 4.

During these research, we'll see the state of the art of TRIZ for Software and the work that has been done until now. Some authors claim it can be used for this discipline, but few suggest where.

We will apply it to Software Architectures. A possible Systematic Model is proposed and described in section 5. It includes some TRIZ tools and it's integrated with other to help architects in Alternative Generation and Evaluation.

This model has been tested in a real-world system, during its development. Section 6 is a summary of the application case, the steps followed by architects, the observed results and the experiences obtained during the process. A final result of the Software Architecture is illustrated and presented on this section too.

The inception of these method as a possible approach to cover the need of a Systematic Method for Software Architectures and the results obtained after its first test lead us to new questions that need to be solved.

2. State of the Art

Is there anything like TRIZ for Software? Is it possible to systematically apply innovation into Software Development?

Software has been considered a creative process, because most of time it's related to new product development. It is: new problems, new solutions, new code many times. Although engineers have tools and methods, and they use them very often, something must be changed to achieve the solution. Software development is then related to the artistic creation, just like music: same instruments, same symbols, but the product is different every time.

Innovation is needed in software, because of the challenges the industry is facing: systems growing in complexity and functionality, more source code, which means a more difficult maintenance. The objectives are to completely fulfill customer needs. The challenge is maximize the useful functions and not the resources.

Much of the research related to TRIZ for software focuses on that approach. The work of some authors is concerned to these statements, as it will be seen in the next sections.

2.1 Some TRIZ Theory

2.1.1 What is TRIZ?

TRIZ is a problem solving method based on logic and data, to solve problems creatively. "TRIZ" is the Russian acronym for the "Theory of Inventive Problem Solving." G.S. Altshuller and his colleagues in the former U.S.S.R. developed the method between 1946 and 1985. **[1]**

TRIZ research began with the hypothesis that there are universal principles of creativity that are the basis for creative innovations that advance technology.

From general problems you can find a general solution and then apply it to a particular problem. There are some methods used in TRIZ; the 40 Inventive principles of problem solving is the most accessible tool.

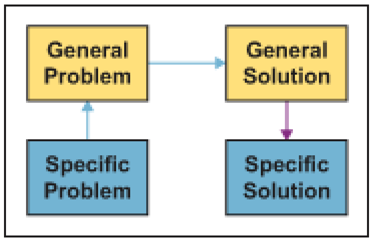

Figure 1: TRIZ problem solving method

TRIZ research began with the hypothesis that there are universal principles of creativity that are the basis for creative innovations that advance technology. If these principles could be identified and codified, they could be taught to people to make the process of creativity more predictable. The short version of this is:

Somebody someplace has already solved this problem (or one very similar to it.) Creativity is now finding that solution and adapting it to this particular problem.

Altshuller and his colleagues demonstrated this by analyzing more than 2'000,000 patents in the former USSR, finding that the same principles can be applied in different fields. This approach has been used too in fields that, apparently, TRIZ cannot be used. In Software, Darrell Mann has analyzed about 40,000 software patents to find the correct forms of the 40 inventive principles in this field.

A fundamental concept of TRIZ is that contradictions should be eliminated. A contradiction is a situation when we want to introduce a change into a system to improve, but something else in the system prevents to reach the desired state. In other words: when something gets better, something else gets worse. Contradiction may be classified in two categories: Technical contradictions and Physical contradictions. **[1]**

2.1.2. TRIZ Tools.

TRIZ has some tools that helps in the Problem Solving:

- 40 inventive principles.
- Contradiction Matrix.
- Su-Field Analysis.
- ARIZ.
- 76 standard solutions.

First three on the list will be covered in this research.

Su-Field Analysis.

Substance-field analysis allows the creation of a model that is representative of the system under discussion. The application of these principles is extremely powerful in defeating psychological inertia and increasing the innovative level of the solution (increasing the level of idealism as well).

In order to understand this tool, the article written by Michael S. Slochum **[12]** is very straightforward. It was read just to know how the tool is used in traditional TRIZ and why Kevin C. Rea considered it for applying this method to Software Development.

The concept of System is very similar in all fields and the plain and simple elements of completeness and incompleteness give a perfect view of how ti see a new system in the very early stage of conception.

40 inventive principles and the Contradiction Matrix.

The 40 inventive principles and the contradiction matrix are the most accessible tools.

The Contradiction Matrix is a database of known solutions (principles) able to overcome contradictions.

When you have a system property or characteristic that you want to change or improve, you may find something that prevents it. I.e, You need a static object to be longer without becoming heavier. This is a contradiction. The improving feature is 'length of stationary object ' and the worsening factor is 'weight of stationary object '. The matrix is used then to discover the possible principles to overcome these contradictions (the principles).

The 40 inventive principles were developed to be applied in mechanics, physics and chemical fields. They've been adapted to business and social fields and also has been considered for Software

Development, as seen in the work done by Darrell Mann **[4][5]**, Kevin C. Rea **[2][3][11]** and Ron Fulbright **[10]**.

2.2 Innovation.

What is innovation? What characteristics has some idea to be an innovation? What levels of innovation do exist?

Some authors state that an innovation is a new idea that becomes accepted, used and a marketing success, bringing revenue to the company **[5]**. This is not the most correct definition for Innovation, because, in software, there have been many successful products that weren't innovations (Microsoft Internet Explorer, as an example).

Innovation implies paradigm shift in some cases, a totally new way to do a task, perform an action or process or a tool, but these cases are only the Macro-Level Innovations, which are only 1% of all.

Innovation can too be slight changes into a previous idea. Changes are gradually added to a system, until the Ideal System is produced. Innovation is then a discontinuous jumps towards a more ideal system.

Where innovation comes from? New ideas don't come out easily. An Innovation process is needed in order to get a new product. Test and fail processes are only effective when you can make enough (and enough can be thousands) tests. Edison was able to produce a lot of innovations by test and fail processes, but he was assisted by a lot of helpers that can perform many of those tests concurrently, but for a sole individual it's impossible to make enough tests for a product to work.

When there's no prior information and experience, test and fail can be the only way. Information an experience, many times, is understood to be registered in the same Knowledge Area. People, most of time, is very committed to their Area and cannot accept or use the work made in another Area. So, discoveries made in Chemistry would not be used in Mechanics and so on. By the way, it's thought that none of the practices in physics, chemical and medicine areas is applicable to Software.

The message behind this is that there's no need to reinvent the wheel each time. There's already a lot of knowledge and work done. The same principles can be applied when the problem is similar, the application may be different. Get a general solution to find a particular solution.

Innovation is, very often, achieved by analyzing a past event or discovery with a methodology. As an example, Toru Nakagawa writes about the development of the Structured Paradigm in programming.

When Dijkstra introduced Structured Programming Paradigm, the GOTO programming was widely used in software development. The debate was long and hard. GOTOers argued that three control structures weren't enough to write a complete and functional program, while the group in the other hand argued GOTO programming was inefficient, hard to maintain and repetitive and with the three control structures, proposed by them, software development would gain benefits.

GOTOers claimed they need the GOTO sentence in order to get all they functionality they needed, and the three control structures proposed by Dijkstra weren't enough for them. Structured Programming was accepted after two new control structures were introduced, completing the ideal system. In that moment, it became a successful innovation, and it's a few of the Macro-Level Innovations until now.

Nakagawa reviewed the Structured Programming proposal using the TRIZ principles 1, 6 and 7, it was found that system was incomplete and that by applying the inventive principles of Nesting, Universality, Segmentation and Trimming, the same solution would be achieved. **[6]**

One of the most important aspects of this article, and one of the main ideas for this research, is the viewpoint: analyze past and solved problems and find how would they be solved using TRIZ principles. Same approach used by Altshuller and Mann, as discussed in section 2.1.

Darrell Mann too supports the approach of finding principles for developing an innovation process by following the same method used by Astshuller. He himself has analyzed over 40,000 software patents in order to set the 40 principles most suitable for Software. **[5]**

2.3 TRIZ for Software

I want to start this section with an overwhelming situation:

If a software engineer would go to his boss and tells him "Boss, I can make our software work 5 % faster". His boss probably replies, "That's of no importance, next month we will get the new processor which works twice as fast. Get back to your work, you still have 20 bugs to solve before the end of this week and don't waste any more time". **[13]**

This is the most clear example that states the Software Development situation: finding solutions that satisfy customer needs instead of improve performance or resource usage. TRIZ for software must focus not in physical contradictions, but in solving contradictions that avoid the successful achievement of the specified needs.

User attention in nowadays software is focused in getting more and more functionality. Processing power and Hardware limitations are no more problems since Moore's law has proved to be true. Users want more from software to use it on his Personal Computers, that have become more powerful. This, turns software into a very complex entity. Challenges are to add new functions in existing software without affecting the existing, creating new and complete solutions keeping the system simple for change. Software is becoming very complex.

Simplicity and reducing complexity are two goals when a system grows. But designers and problem solvers must define what is Simple first. The debate among the GOTO paradigm and the structured programming, as discussed in previous section, offers a good view of how hard is to define Simplicity.

Software engineers must have the skill to read symbols on paper and imagine what the final result will be. Something that musicians can do easily by reading a musical score and producing a mental musical listening.

Most of the software is related to linguistics: rules, grammar, symbols. So, in order to give a solution to a problem with software, semantics is important to write the algorithms. **[7]**

2.3.1. The 40 Principles and the Contradiction Matrix

Genrich Altshuller developed the 40 Principles more than 20 years ago. He found utilizing principles previously used to solve similar problems in other inventive solutions could solve technical problems.

These 40 Principles have a way to be applied to Software to solve some problems, but the only listed are particular and small problems or small pieces of code.

Some authors state the first approach to bring TRIZ into Software came from Kevin C. Rea researches. Not clear nor rich in content, but it states that exist some works related to TRIZ for Software. Examples are short and covers a wide variety of IT areas. There's at least one example and analogy for each Principle, but it's only stated that each principle and analogy can be applied in Software, it doesn't explain anything about how.

The author makes clear that, at the time he wrote the article, there were very few work in IT using TRIZ; that's the reason why some Principles didn't have an example. **[2].**

Later, Ron Fulbright completed Rea's 40 principles analogies for software, by adding software-related analogies for the principles 29, 31, 36, 37, 38 and 39. There weren't any analogies before and it was thought there weren't any relationship between these principles and their physics application and software.

Fulbright completes these matrix and provides a table with them and some examples, but not detailed, the same as Rea. **[10]**

Kevin C. Rea promises to give a practical example of the 40 principles use in software development, but first, he offers an apologize for not giving an applied-true example because he has some legal restriction for doing this.

Instead, he "invents" a new example to show us how to apply the original 40 principles directly on software, but it is not a real software solution, but a management and process improvement.

I think he has more real examples used in Software stages rather than the process of bridging the gap of communication among team members and developers. **[11]**

2.3.2. Concurrent Systems and TRIZ

Many software problems are caused by a test and fail programming. An inventive way of finding solution could provide solutions for the challenges of the 21st Century.

Applying TRIZ in the software-problem domain has many potential benefits. With it, algorithmic templates and enhancement of TRIZ tools (like su-field analysis) can be achieved.

In **[3]**, Kevin C. Rea introduces the concept o Metapattern. The introduction of a Metapattern is a new approach to conceptual information modeling. With the metapattern, analysis is recognized as a critical activity in its own right during information system development. The metapattern is not a method for technical design or software engineering; it is a highly focused analysis tool.

Also, Context-templates (meta-template) serve to complement TRIZ in the context of software problems as well as other thinking methods.

Rea's proposal of Metapattern and Metamodel tries to bridge the gap between academia and industry. Without a detailed example or solutions, it offers better views about TRIZ applied to Software Engineering. Rea explained in his own works later than he cannot publish a detailed example, because of some legal restrictions acquired in his former job. There were very few clues of these until now, with a concrete use of TRIZ in Software Design.

However, the work presented here is merely theoretical. There's much to do yet in the words of the author before presenting results in applying these approach in Software Engineering. It seems a good idea, but it lacks support given by a case. **[3]**

According to Rea, Su-Field and context templates can bridge the gap between academia and practice by defining common solutions to problems. He proposes the use of su-field analysis in designing the software solution. Darrell Mann writes a criticism about the su-field tool. It is stated, but not demonstrated, that that step is too far for most software engineers. This put us again in darkness, because there's so few information about how to apply a particular tool to solving software process.

However, it's stated that only with the 40 principles and the contradiction matrix most of the software problems can be solved, but using it in a different form than the original TRIZ. **[4]**

Darrell Mann states that's too much engineers; steps too far for them and the contradiction principles can be enough to achieve innovation. **[4]**

2.3.3. Software Architectures

At first glance TRIZ doesn't seem to apply to software problems: no atoms, no molecules, no layers to touch; no physical, no chemical effect to apply. Yet software problems have been successfully solved with the use of TRIZ. But, according to many, still a lot of work needs to be done.

Software development is more concerned to manage the increasing complexity of new systems rather than making faster algorithms and dealing with limited resources. The challenge nowadays is to manage this complexity.

Another way to deal with the increasing complexity is to create architecture of the software. Software architecture provides the technical structure for a project. A good architecture makes the rest of the work easy. A bad architecture makes the rest of the work almost impossible. **[13]**

Thus, software development finds the best solutions in Design Phases. That is, the solutions that can achieve both, the user-valuable functions and the capability to modify software and add more functions and improve the existent.

How does an engineer choose the best design? David Warburton formulate the question in **[8]**. We don't have the answer, that's why we have such difficulties and darkness in software industry.

Although Warburton doesn't offer a Silver Bullet, he explains some of the causes for flaws in final products: concept evaluation and selection of the best design that solves a particular problem.

The development of a new medical device is not as helpful as an example for the software matters, but the ideas about requirement priorities, design generation and refining are very similar in both fields. **[8]**

Much of a product success is dependent on the concept phase.

The performance of the development team ultimately depends on the design skills of the individual engineers. Many engineers keep responding to the test failures with further refinements because they are, at some point, psychologically committed to seeing a particular design succeed.

The Pugh concept selection matrix provides a technique for choosing among design alternatives. This won't be explained here, because it will be assessed later as part of further research.

2.4. The ISQ

The use of ISQ (Innovation Situation Questionnaire) in TRIZ provides a preliminary context of the problem, because to give a solution, the first thing you need is a problem definition.

The original approach of ISQ is to non software related problems and it's taken in order to know the actual situation of the system you want to improve or the problem you want to solve, but its structure is very similar to the Use Case Templates proposed and used in software engineering. Both provide the initial information for further work and analysis. **[9]**

The authors in **[9]** don't write about software and I think they're not aware of it can be applied to Software Deveolpment as Use Cases. They focus their work on System's primary useful function, an area where TRIZ can find contradictions, harmful functions and how to achieve the primary function. Use Cases are detailed documents about processes or Elemental Business Process and they can be seen as Primary Functions in the Software System to be developed.

The structure of the ISQ is:

1. Information about the system you want to improve/create and its environment.
1.1 System Name
1.2 System primary useful function.
1.3 Current or desired system structure.
1.4 Functioning of the system.
1.5 System Environment.
2. Available Resources.
3. Information about the problem situation.
3.1 Desired improvement to the system or drawback you want to eliminate.
3.2 Mechanism which causes the drawback to occur, if it's clear.
3.3 History of the development of the problem.
3.4 Other problems to be solved.
4. Changing the system.
4.1 Allowable changes to the system.
4.2 Limitations to changing the system.
5. Criteria for selecting solution concepts.
6. History of attempted solutions of the problem.

Table 1: ISQ Structure

We can observe a slight likeness between the ISQ and the Use Cases Template. We discuss it more detailed in section 3. Both artifacts gather the same information: what the system will be used for and the current situation of it.

This was a very opportunistic finding, because of the source is very apart from software bibliography. It let us know that TRIZ principles are true, knowledge can be useful in many different areas because the most important thing is the concept, the principle; the implementation may be different.

Systematic Innovation can be achieved by observing this.

3. Research Report.

In this section we will see an overview of the objectives for future research.

The main focus of TRIZ for software is maximize the Primary Useful Functions. These are discovered and developed in two stages of the Software Life cycle: Requirements and Architectural Design. Selecting a good design in early stages of the life cycle is crucial for the solution.

Software Development, nowadays, implements some artifacts in the first stages of the project life cycle. Also, there's a number of proven and spread methodologies that support the use of those artifacts. Many of them have some variances among the organization or individual using them, but the base is very similar.

The focus stages of the Software Life Cycle to be covered here are Requirements and Architecture.

Some known artifacts for requirements are:

- The Vision and Scope Document.
- Use Cases Template.
- Software Requirements Specification Template.

At the beginning, it was found that the ISQ has some correspondence in Use Cases Templates, where the Elementary Business Processes (or User Requirements) are modeled. The ISQ structure is reviewed in Section 4.

Use Case Template structure, according to that proposed by Alistair Cockburn (**Figure 2**), shares some likeness to the ISQ and stores very similar information about the problem domain. ISQ is intended to find the System's Primary Useful Function and all the Environment information about the system to be created or improved. Use cases are intended to discover the User Needs and much, not all, information about the process environment and third-party influences (business rules, other systems).

1. Learn to fill in all the fields of the template in several passes, at several moments in the project's requirements gathering and project setup work. Here is a sample sequence. First, fill in just these fields, for all the use cases you need to consider at this time:

Use Case: <number> <the name should be the goal as a short active verb phrase>
Goal in Context: <a longer statement of the goal, if needed>
Scope: <what system is being considered black-box under design>
Level: <one of: Summary, Primary task, Subfunction>
Primary Actor: <a role name for the primary actor, or description>
Priority: <how critical to your system / organization>
Frequency: <how often it is expected to happen>

2. Stare at what you have so far. Think. Examine. Can you merge or remove some of them? Can you partition them into ones that should be developed together, or written later? For the ones you determine to pursue now, fill in the following fields:

Trigger: <the action upon the system that starts the use case, may be time event>
MAIN SUCCESS SCENARIO

3. Now you have enough information to check your project's scope and look for surprises. **Before you are done describing the system's functioning,** you have to fill out:

EXTENSIONS
SUB-VARIATIONS
Superordinate Use Case: <optional, name of use case that includes this one>
Subordinate Use Cases: <optional, depending on tools, links to sub.use cases>

4. You now have the system's functionality captured. **When you are ready to work on your estimations,** fill in:

Performance Target: <the amount of time this use case should take>
OPEN ISSUES
SCHEDULE

5. Finally, when you are in the final stages of project estimating, you need to **identify all the systems to which you will have to build interfaces.** Fill in:

Channel to primary actor: <e.g. interactive, static files, database>
Secondary Actors: <list of other systems needed to accomplish use case>
Channel to Secondary Actors: <e.g. interactive, static, file, database, timeout>

Figure 2: Use Case Template ***[14]***

Use case describes how the system to be build will be used and, in many cases, what for. However, it describes the needs of a single user or actor.

and Pugh helps by giving priorities to the requirements and identifying the Utility Function (with a given formula). The Utility Function is similar to the TRIZ concept: Primary Useful Function.

The Problem Forming and the Problem Solving

Software development is about problem solving and automation of processes. Requirements Elicitation and analysis are important stages before solution construction. Information is crucial in early stages and software engineers must create useful artifacts containing it and use them in next stages, according to a methodology as inputs. Use Cases are one of the most used artifacts for requirements analysis and after comparing with ISQ they can be used as input for some TRIZ tools.

This research will focus in finding what TRIZ Tools can be used in Software Development; the Software Product Life Cycle Phases selected for this starts with Requirements Analysis and Specification (Artifacts: Use Cases, Vision and Scope and SRS), Software Design (Architectures and High-Level Design).

Problem forming and definition will be covered within the Requirements Phase. ISQ will be replaced with Requirements artifacts, as entry for Architecture and High Level Design. Those designs will be proposed first, then analyzed with a TRIZ approach and implementing TRIZ Tools.

The goals are:

- Use the existent artifacts for Software Requirements process as is and evaluate if they can be used as an entry for the 40 inventive principles and contradiction matrix in Architecture Selection.
- The strategy is to use an Iterative Method. I mean: no full-documented requirements, but a subset of them. Software Architecture will be based on Vision and Scope document and High Level Design on Use Cases subset.
- Apply the 40 inventive principles (as proposed by Rea and Mann) and Pugh Method for High Level Design in a business application. Transactions and rules interacting in business processes, described and documented in Use Cases. The level of concurrency for this systems is expected to be low. The platform is World Wide Web, applications are not open, but used as Intranet operative applications.

Su-field analysis will be excluded, unless the Metapattern and Metatemplate would be released and explained. There's no reason why to use them or implement with the current information.

4. Software Inception with TRIZ

Once the problem has been formed in single statements, the next step is to think how to solve it. There are many methods and frameworks defined for problem solution: TRIZ (our subject of study) and the Polya Method "How to solve it?" as some examples.

It has been observed that one of the most used strategies in software development is test and fail. The formula is simple: write some code, test, fix, write some more code, test, fix, write more code, test, fix... and repeat, maybe, forever. Unless you have unlimited resources to produce an infinite series of tests, test and fail is the most expensive technique in software development

This approach has also other trade-offs: because the vision of the product the developer has in a given moment limits just to the piece of code that is being written.

Best practices of software development recommend to develop a previous solution (often called Design) that can be evaluated and tested before the product construction. Design provides information for evaluation (even testing) of the solution before its implementation. It allows the designers to think about possible scenarios for the final product and determine if it will behave as expected for the current situation.

The importance of the design is high, so some Development Methods and Quality Process (even those defined for individual, such as PSP) set Design before any implementation phase.

Design must start with the big picture: envisioning of the system as a whole, identifying parts, responsibilities and interactions among parts. At this level of detail, it is described what the system does or can do, but not how to do it. Just establishes the principles where the subsequent designs will settle on.

In the next sections, I describe the results of the study. First, I'll describe some needs and difficulties found in Architectural Design process. Then, I'll tell how the combination of existent tools can work as an Assistant Method for architects. Finally, I'll show the results of a pilot application of this method in a system development.

4.1 The need for architectural design.

A software architecture is developed as the first step toward designing a system that has a collection of desired properties **[15]**. It is defined as The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them.

Software architecture deals with the design and implementation of the high-level structure of the software. It is the result of assembling a certain number of architectural elements in some well-chosen forms to satisfy the major functionality and performance requirements of the system, as well as some other, non-functional requirements such as reliability, scalability, portability, and availability [18].

So, as a first step, an individual or team (architects) develops an architecture, evaluates it and ensures its correct implementation. Simple, isn't it?

Well, it isn't. A Software Architecture cannot be made by anyone, and not many organizations include this activity into their process.

Why?

Software Architecture is influenced by more than Functional Requirements. It is influenced by the System Stakeholders, whom have different concerns that they wish the system to guarantee or optimize **[15]**. Some of these influences are called Quality Attributes such as performance, reliability, availability, platform compatibility, memory utilization, network usage, security, modifiability, usability, and interoperability with other systems.

Architectures is also influenced by the architect's experience. If the architects for a system have had good results using a particular architectural approach, such as distributed objects or implicit invocation, chances are that they will try that same approach on a new development effort. Conversely, if their prior experience with this approach was disastrous, the architects may be reluctant to try it again. Architectural choices may also come from an architect's education and training, exposure to successful architectural patterns, or exposure to systems that have worked particularly poorly or particularly well. The architects may also wish to experiment with an architectural pattern or technique learned from a book or a course. **[15]**

There are few systematic methods and tools for helping engineers in creating an architecture, also there are few or none methods for architect training and forming. Architects, usually, are very experienced developers that can envision a solution based on their prior experiences. Becoming an architect is a long and time consuming journey. Candidates must gain expertise in programming and designing software solutions. Experience is the most used criteria for Architecture Selection and Design. New Architects must be working with a Senior Architect, a very time and resource consuming approach.

4.3 The problem with Architectural Design.

Teams and organizations faces difficulties in forming new architects. By working with senior architects for a reasonable amount of time, they will learn to think the same way their mentors do and will know their particular scenarios and systems.

New architects must learn to solve problems related to Software and Systems. Some problems can have more than one valid solution alternative. Each alternative has its pros and cons and the combination of them gives us the one that's more ideal. After finding the alternatives, the new architect must evaluate them and select the one that best solves the problem.

There are, then, two major activities a new architect must learn:

- Create architecture alternatives.
- Evaluate the alternatives.

Each activity can be done by following formal or empirical methods. Can both be performed systematically?

4.3.1 Generating Alternatives.

If we see the process backwards, starting with the Evaluation of Alternatives step, we can find some methods for evaluation and comparison.

Methods for evaluating alternatives receives two entries: a weighed criteria list and a list of alternatives. Depending on the method, there are more or less steps, but all of them share the core of comparing alternatives with the given criteria to find the one that best fits in it.

The methods and models studied (Pugh, CMMI, the Evaporating Cloud) tells how to evaluate alternatives, but very few suggest how to generate them. Architects must generate one or more alternatives before they can evaluate them.

As Kazman, Clements and Bass wrote in **[15]**, architects will apply the patterns they've learned with time. Their experience will dictate them what decisions to take and what path they can follow. For some of the new architects, their background experience is related only to development and implementation. Often, they will make decisions based on technologies they already know or have worked with.

Architecture literature suggest some tools that can be used as a base to start generating alternatives. These tools are the architectural styles, and many books and articles describe how and when to use. Styles support a particular quality attribute and, as we saw before, systems must adhere to many attributes at the same time. Also, the set of different styles may overwhelm the architect.

We are proposing the use of TRIZ for Alternative Generation. Ellen Domb**[16]** supports the idea of using the 40 principles for alternative generation, because some researches indicates that inventors using TRIZ experience improvement of 70% to 300% or more in th number of creative ideas that they generate.

The 40 principles offers a structured approach to the generation of ideas. They lead to the conflict and offers an principle that can solve it. We will use the principles in order to guide the architect to detect a particular situation and try to find a style that can solve it.

4.3.2 Alternative Evaluation

How to make a choice?

The common steps involve the evaluation of alternatives against some criteria and select that alternative best ranked after evaluation. The better your Evaluation Process, the better choice you can make.

Evaluation Process must be consistent along time and must be individual independent. Across an organization, the evaluation of alternatives, performed by different people, should have similar results using the same method. Organizations can rely on Experts judgment, but must not be dependent of it.

Evaluation methods should help teams and individuals to provide a rationale of their decision. The evaluation can be sometimes a numeric value or a boolean value. Each technique is performed in a different way. Boolean techniques may involve possible scenarios, answers as “yes” or “no”, and evaluates if an alternative passes the scenario. Numerical techniques may be to assign a grade using a scale (0 to 10, by example), according to evaluators perspective.

The objective on this study is to use a weighed numerical method as a formal alternative in order to reduce subjectivity in architecture alternative evaluation.

5. The Junior Architect Assistant

The Junior Architect Assistant is the application of TRIZ, the Pugh Selection Matrix and Software Development artifacts, in order to define a Systematic Method to create, evaluate and select an architecture.

The idea behind the Assistant is to guide junior architects to take his own decisions, based on the information he posses in a moment, and can provide the rationale to generate solution alternatives. By using the tools, the Junior Architect can isolate the concepts contained in the situation he is analyzing and generate alternatives for that characteristics mixture. The architect would avoid repetition of patterns, using all over again the same solutions that were effective for past problems, even if there are more ideal alternatives.

The model can also teach the architect to think twice and prevent of taking one alternative that he used before and worked. The architect must look to evaluations made previously and compare them with the current situation. Evaluation Criteria is not the same for every system, even when criteria names are the same, priorities may vary and the decision to take at that moment should satisfy the new prioritized list.

The Junior Architect Assistant can help a team or an organization to achieve some of the goals stated in CMMI Process Area Technical Solution (TS):

- SG 1 Select Product Component Solutions
- SP 1.1 Develop Alternative Solutions and Selection Criteria
- SP 1.2 Select Product Component Solutions
- SG 2 Develop the Design
- SP 2.1 Design the Product or Product Component

5.1 Used Tools

- Requirements Artifacts: Vision and Scope, SRS, Use Cases.
  V&S provides enough information to obtain the Primary Useful function of the system, also with some Quality Attributes data. SRS and Use Cases provides scenarios for evaluation steps and some information to identify the utility functions.
- TRIZ 40 Principles and the contradiction matrix.
- Evaporating Cloud.
- Pugh Concept Selection Matrix.

5.2 The Method

The Junior Architect Assistant defines a series of steps to aid architects during the Architecture Design process. Each step uses one or more of the tools listed in section 5.1, in order to generate the products for the next phase.

The model is presented as a Flow Diagram in Figure 3.

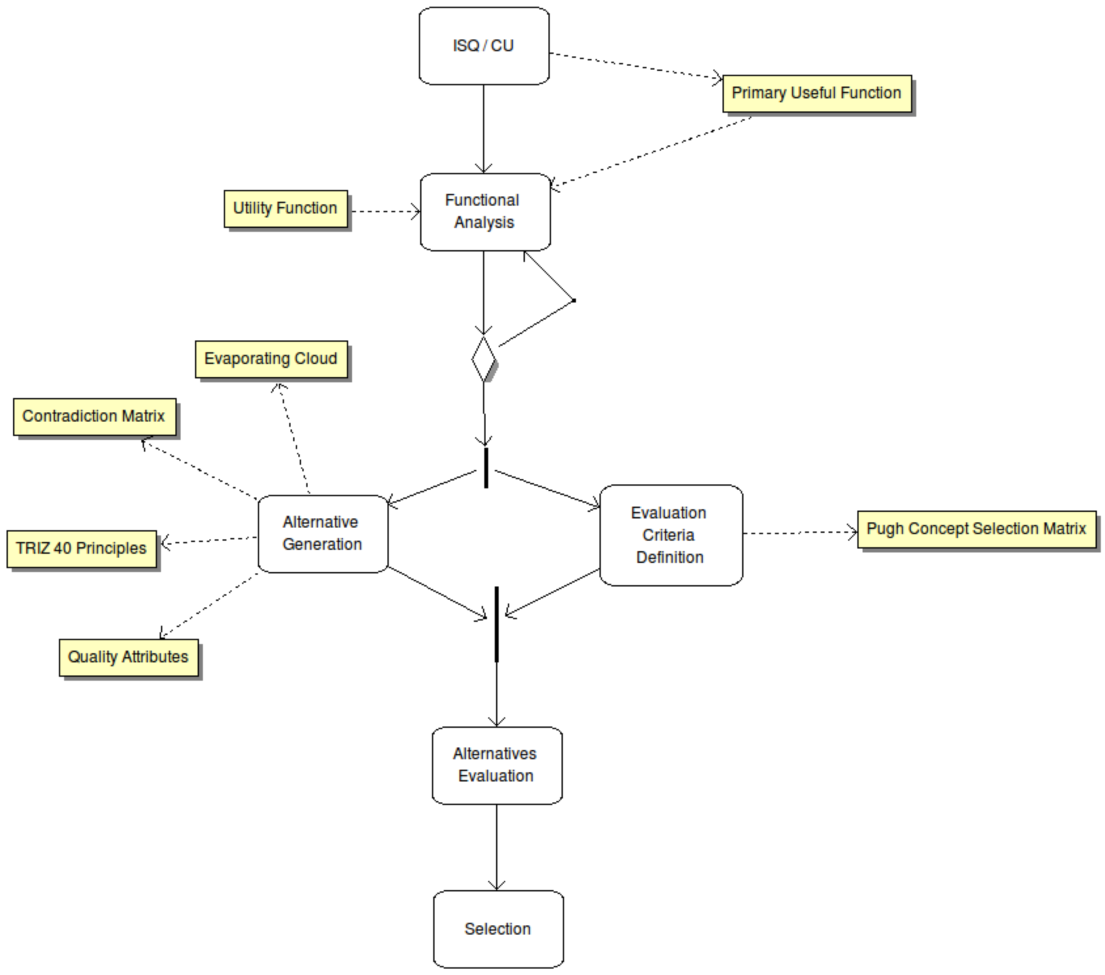

Figure 3: The Junior Architect Assistant flow

The guide script is depicted in Table 2

The Junior Architect Assistant		
Purpose	To provide guidance during Architecture Design Process.	
Entry Criteria	• Documentation containing the business goals and requirements. ◦ Vision and Scope ◦ SRS	
Step	**Description**	**Products Generated**
1. Define the Current Situation.	• Analyze the Requirements Documentation and list the main features that the system must achieve. • Find or gather the Quality Attributes that the client may be concerned of. • Find or gather the Goals and Quality Attributes the development organization management may be concerned of. • List the Quality Attributes in order of priority.	• List of System Features • List of constraints and quality attributes
2. Functional Analysis	• Decompose the features into pieces to get the responsibilities of the system. • Repeat this step until there's no more identifiable characteristics or responsibilities.	• List of System Responsibilities.
3. Alternative Generation.	• Find the parts (objects) of the system that will fulfill each of the responsibilities identified in step 2. • Find the relationship among components, in order to establish dependencies and collaboration. • Use the 40 principles as a • Develop the Class and Component diagrams for the solution alternatives found by the team.	• Solution alternatives documented.
4. Prepare Evaluation Criteria	• List the Quality Attributes and evaluation criteria for the current system. • Assign a weight for each of the • Produce the Concept Selection Matrix with the weighed criteria.	• Pugh Concept Selection Matrix
5. Alternative Evaluation	• List the alternatives in the Selection Matrix. • Evaluate each alternative against criteria. Give 1 if alternative offers advantage in that criteria. 0 if doesn't add any value and -1 if substracts value on that element.	• Pugh Selection Matrix Evaluated.
6. Selection	• Choose the Alternative with the highest	• Alternative

	evaluation in step 5. • Document and justify the selection, using the Selection Matrix.	documentation.
Exit Criteria	• Solution alternative	

Table 2: Junior Architect Assistant Script

5.3 Support background and literature.

The model is based on CMMI Process Areas Decision Analysis and Resolution (DAR) and Technical Solution (TS). **[20]**

DAR recommends to follow these guidelines:

- Establishing the criteria for evaluating alternatives.
- Identifying alternative solutions.
- Selecting methods for evaluating alternatives.
- Evaluating the alternative solutions using the established criteria and methods.
- Selecting recommended solutions from the alternatives based on the evaluation criteria.

DAR also recommends the use of a formal evaluation process, because it reduces the subjective nature of the decision and has a higher probability of selecting a solution that meets the multiple demands of relevant stakeholders. Methods based on subjects judgment were discarded for the Assistant.

Pugh Concept Selection Matrix was selected because of its easiness of use and flexibility. The evaluation method (1, 0, -1) is used as described in **[23]**. Some other papers suggest a graphic style for the matrix. I selected these because it can be used straightforward with Spreadsheet software.

Recommended criteria from TS was used as aid for evaluation criteria preparation. Recommendations of the model are as follows:

- Cost of development, manufacturing, procurement, maintenance, and support, etc.
- Performance.
- Complexity of the product component and product-related lifecycle processes.
- Robustness to product operating and use conditions, operating modes, environments, and variations in product-related lifecycle processes.
- Product expansion and growth.
- Technology limitations.
- Sensitivity to construction methods and materials.
- Risk.
- Evolution of requirements and technology.
- Disposal.
- Capabilities and limitations of end users and operators.
- Characteristics of COTS products.

This list is a recommendation only; some criteria may not apply to a particular system and new criteria may be inserted in the Pugh Concept Selection Matrix. The weight of each evaluation criteria must be defined by the team.

5.4 Advantages

- Lightweight.
- Supported with tools.
- Self-feedback and decision database. A recommended alternative is accompanied by documentation of the selected methods, criteria, alternatives, and rationale for the recommendation. The documentation is distributed to relevant stakeholders; it provides a record of the formal evaluation process and rationale that are useful to other projects that encounter a similar issue.
- Based on practices and recommendations of World-Class Quality Models, such as CMMI.

5.5 Why Architecture Phase?

Herman Hartmann, Ad Vermeulen, Martine van Beers **[13]** and David Warburton **[8]**, many of the problems in product development come from a bad concept selection y design phases. Although there are more than one design phase during development lifecycle, Architecture settles the base for the future development and decision making. With the wrong architecture, technical difficulties may appear or those identified at the beginning can get worse.

I observed in a partner organization, during its CMMI for Level 3 Assessment preparation, that using a method similar to the one proposed here in phases where the level of detail (such as HLD or DLD) here adds a lot of overhead, latency and paperwork. It is too much, as Darrell Mann stated in **[4]**, and such phases can be better solved using another tools. Design Patterns are a good enough for engineers in DLD, as an example.

6. Application Case

This model was tested in a small software development organization during the development of one of their systems.

Project was already in implementation phase and some components were already coded and tested. However, very few design documentation was created. The architecture the implementation team was following was based on a previous system design and principal decisions were taking strongly influenced by technological reasons (the organization used a development framework, with many agile implementation functions that accelerated development). This design was taken as the Base design. New components (those not contained in the other system) were designed during DLD.

There were two technical leaders that supervised the implementation and took design decisions at HLD. Both were invited to develop the architectural design of the application they already knew.

The existent requirements documentation was in a compatible format than the one needed for the method to work.

One of the organization's manager was invited for developing the list of Evaluation Criteria. It was used as a first draft for the Pugh Selection Matrix.

6.1 The system.

The system was a Database Transactional and Client-Server based. It had a legacy Database, that must be used in order to keep previous records.

The main purpose for this system was to manage information related to Personal Wealth Reports. Workers must provide certain information about the goods and assets they own and also about their jobs. These reports are then revised, evaluated and approved or rejected, depending on the reigning rules for the current year.

The amount and type of information each user must register in the system also varies. Some workers must register more and some must register less. This is based on a series of rules that are published on January each year.

6.1.1 Functionality.

The final software product was expected to have the features:

1. Allow the user to register his/her personal report.
2. Select and present the user those fields that are really needed. The client wanted to reduce their paper . This is described in Appendix A
3. Reducing the revision time, by filtering and detecting incomplete reports and showing the reviser just the information to be checked.
4. Register and apply revision and selection rules for each year.

As expressed by the customer, the system must allow a worker to register information according to the year and allow an available report checking faster.

6.1.2 Quality Attributes

Customer organization was also concerned for these Quality Attributes.

- Privacy and Security. They were recording very sensitive and private data, that can be only accessed by the organization personnel. It must be hidden and unaccessible for non authorized users and external individuals.
- Usability. The system must be usable for non-technical users.
- Modifiability.

Developing organization has also some goals and concerns for this project. They wanted to take advantage of customer reputation and potential for new projects, and they stated this:

- Schedule. System must be completely functional at a five month deadline.
- Component Reuse. The organization wanted to start a product line strategy and some of the functionality of the system can serve as a base for one or more products. At least, one of the components should be designed for reuse.

6.2 Assumptions and previous preparation

It was assumed that the participants of this case of study were all trained in Design Patterns. Another assumption, too, was all of them understood the concepts of Responsibility in the Software Design Context.

Participants also knew how to use UML and have a little knowledge about Architectural Views in Architecture Documentation.

As an agreement, we called the two participants “Architects”.

Architects were introduced to TRIZ with a short explanation of the method, as illustrated in Figure 1. Then, they had a total of eight hours of training in the understanding of the 40 principles.

6.3 First Design

Once the functionality and quality attributes were defined, the first design was produced. Architects made decisions based on Functionality.

First alternative contained enough modules to cover the set of functions and responsibilities defined as input. Architects used their previous experience in Software Development and designed a solution thinking in implementation: created software components that act exactly as the functions describes.

Two tools (or views) were used for this design: UML Class Diagram and UML Component diagram. Class diagram represent the logical view of the system and helped the architects to identify and assign the responsibilities to system modules, so it will be easier for them to relate each System Function in a “uses” style. Component Diagrams represent the communication among software units to perform a function.

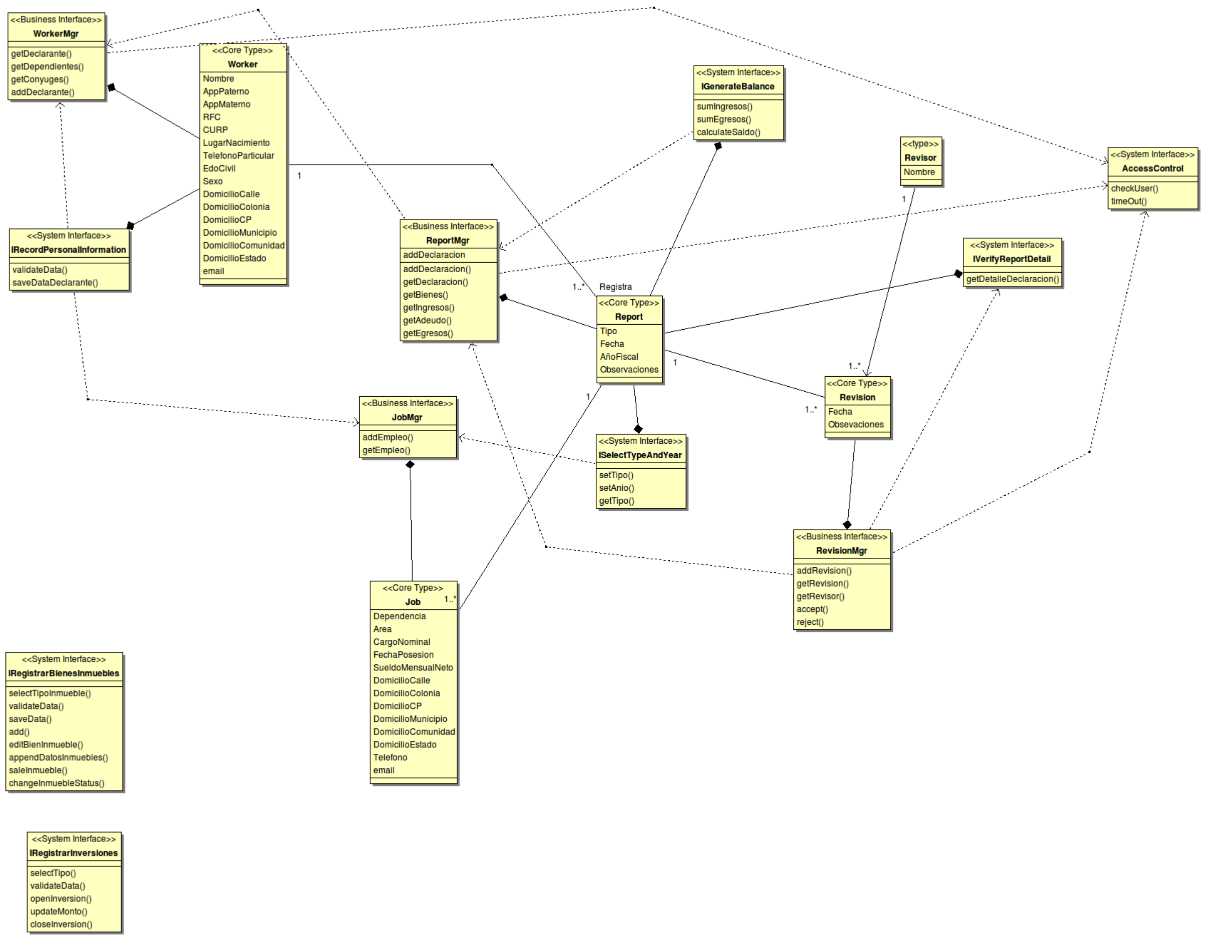

Figure 4: Alternative 1 Object View

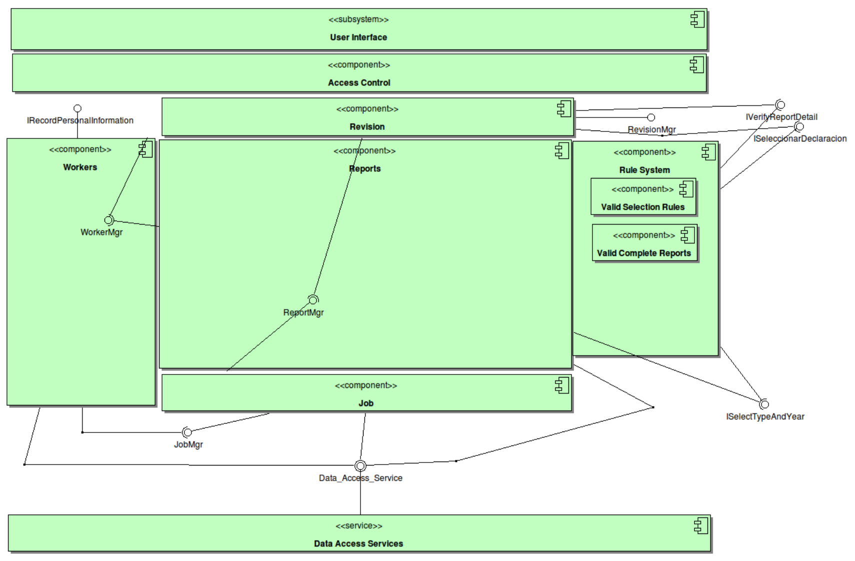

Figure 5: Alternative 2 Component View

After first design presentation, architects where asked to evaluate the alternative, in order to find which quality attributes are achieved with their solution.

The initial draft of the Pugh Concept Selection Matrix, containing the list of evaluation criteria developed with management, was given to the architects, whom use it following a “true” and “false” approach.

Architects should answer each row with a “yes” or “no”; then, according to the results, they had to provide a rationale of why his design achieved or not the system's goals.

These are some of the first evaluation results:

- No reuse. The rule system fulfilled the responsibilities of Report Selection and Report Evaluation, but it was tied to the system's specific rules at the moment the software was defined.
- Duplicity. Two components executed the same function of data evaluation and comparison. The only difference identified by the team was the list of input data for each component. The behavior of each one was practically the same.
- Development take longer. As every validation must be implemented separately, it also must be tested as a unit. The function of data evaluation must be tested more than once.
- Modifiability gets compromised. As new “validation” is proposed or needed or if the current set of rules change, every component must be modified to accomplish the new set of rules.
- The team also found at this point that their alternative didn't satisfied the responsibility 4, because each rule must be defined into the code.

Architects team found there were very little opportunity for reuse in the actual system and design. Many of the implementation was working just for the system's domain. After a review with one of the organization's manager, they defined the component named “Rule System” as an area of interest for component reuse. It wasn't yet implemented.

6.4 Revision and New Alternative Generation

The architects were told to generate one more alternative solution for the system. As an aid, a first draft of the 40 principles mapped to architectural styles was released as a guideline for the architects. This draft was based on the examples of by Rea **[2]** and Fullbright **[10]** for using the principles in software development; then refined with the examples of classic TRIZ in **[16]**. It was presented as a list of each principle name and description, some of the architectural styles that can be used with it and some example of system application.

Architect's goal was to propose one alternative for the feature 4 of the system. The Evaporating Cloud was used to illustrate the contradictions faced by the architects.

The Goal:

- Adaptable system for changing rules.

Requirements:

- Adapt the system to new registered rules.
- Reduce the time of development

Contradictions:

- Modify the code to insert new and active rules.
- Don't modify the code.

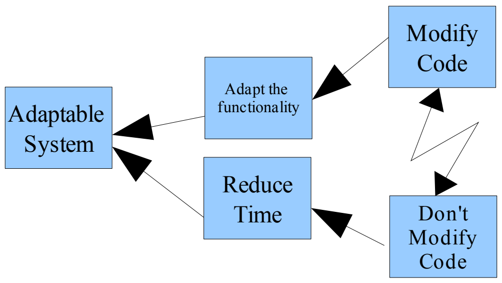

Figure 6: Evaporating Cloud for the Rule System

Architects identified they could solve contradictions by introducing two of the principles:

Self Service: The component configures itself in order to execute the chain of rules.

Segmentation: Validation process was broken into more simple concepts: comparison criteria and data set. Two new responsibilities were identified: “configure the filter chain” and “execute filter”.

Design for this alternative is illustrated in next figures.

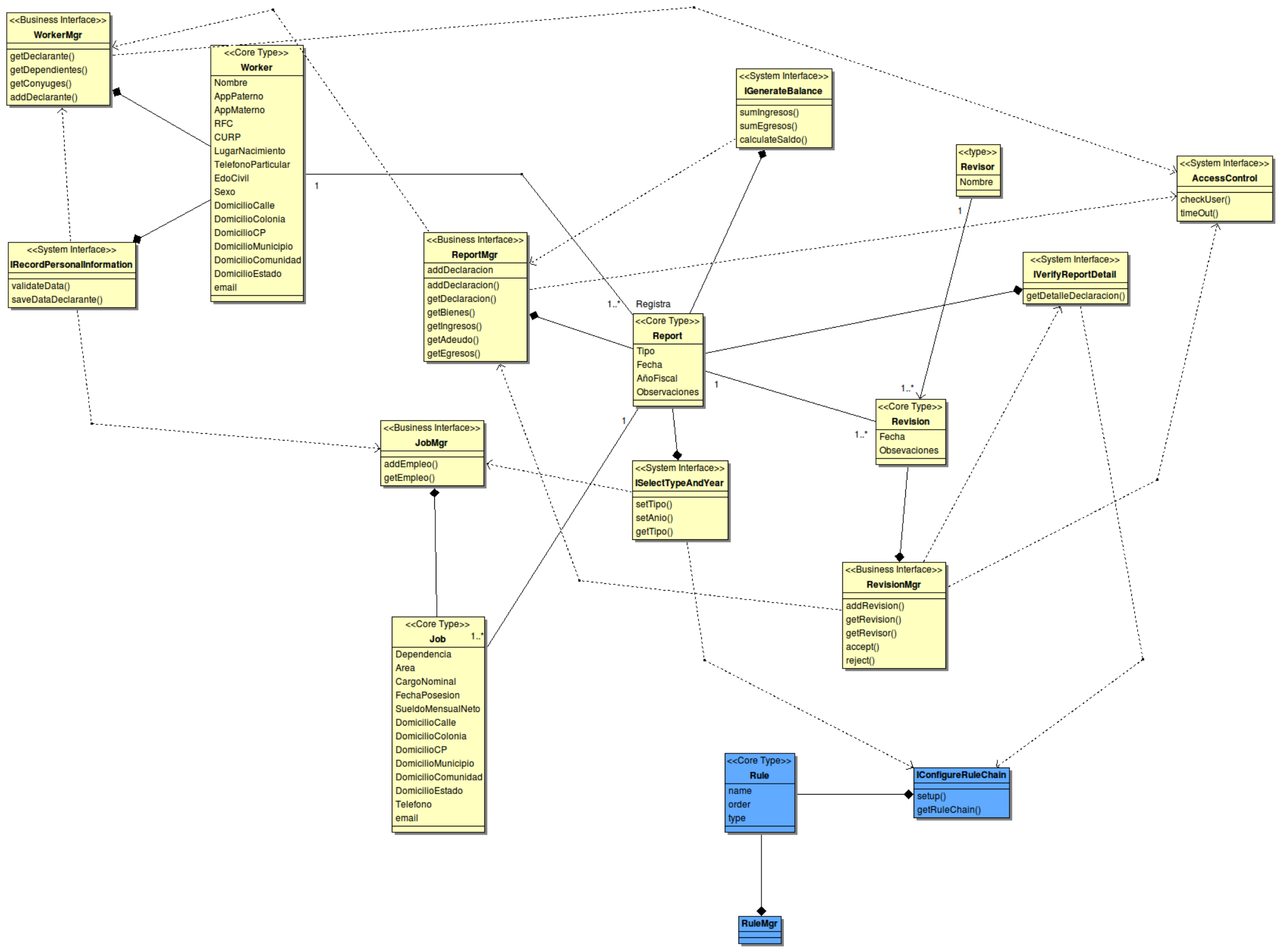

Figure 7: Alternative 2 Object Diagram

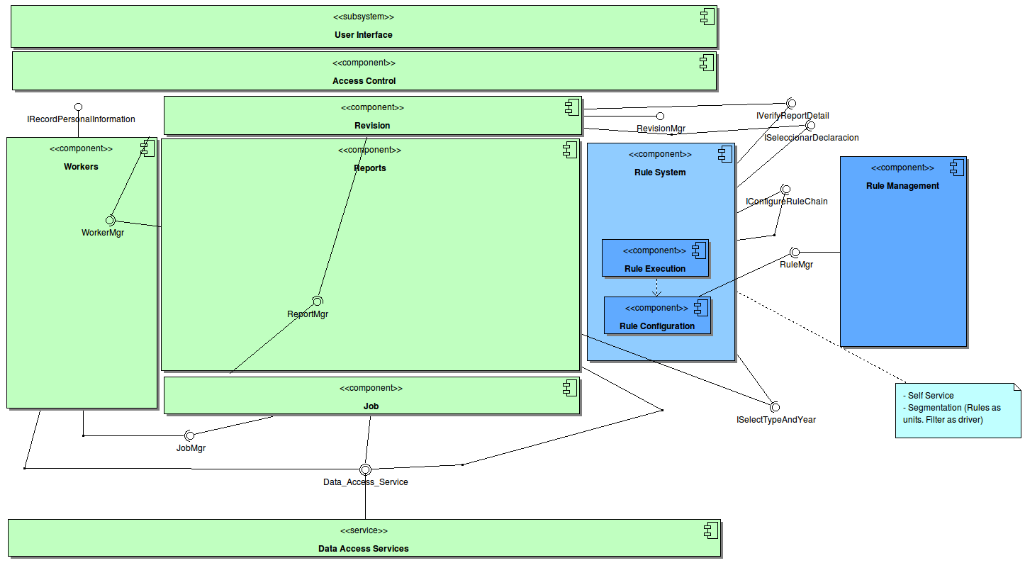

Figure 8: Alternative 2 Component View

6.5 Evaluation and selection.

The two alternatives generated by the architects team were evaluated, according to the criteria established with management.

Weight for each criteria was assigned in order of importance, considering both customer and developing organization concern.

Default alternative was the current system.

	Weight	**Default Choice (Actual System)**	**Alternate #1**	**Alternate #2**
Lower Cost	9	0	0	0
Development time, short schedule	10	0	-1	1
Adaptability to legacy system	7	0	1	1
Product expansion and growth	5	0	0	0
Lower Complexity	6	0	1	0
Reuse	8	0	-1	1
Totals			-5	25

Scale: 0 to 10. 10 as the highest

Table 3: Pugh Concept Selection Matrix for Wealth Report System

Evaluation was made following the method in **[23]**. A three value answer was used to add an extra dimension rather than just use the “true” and “false” approach. I was used that way in order to find the most valuable alternative.

-1 = Disadvantage.

0 = Same as existing system.

1 = Advantage

Alternative 2 exhibited the largest positive benefit of the two alternatives evaluated.

6.6 Results

First design made by the architects showed their limited experience in software development and the strong influence of their current practices and knowledge of technology. They presented difficulties in finding alternatives just basing decision in their knowledge of the code and the framework.

After a quick evaluation of their design, they found they can achieve functionality easily by identifying the modules as Business Units and User Screens, but cannot build those modules for to satisfy criteria different than functionality.

The sequence of steps helped the architects to focus on one activity at a time and isolated concepts for every phase. In new alternative generation, they were thinking in responsibilities separated from technology during the Functional Analysis, they applied a System Point of View and made decisions without being influenced by technology names or tools.

There were difficulties using the 40 principles to architectural style map. Architects found it was very difficult to take an architectural decision using the map, because many of the styles were listed in many principles with no difference between each one. It was easier when they compared the examples with some component or previous experience and scaled it to a macro level.

There was a presentation to management of the selected architecture. Team explained everything concerned to the rule system, the functions it provided and the satisfied criteria for selection, specially that were it offered advantages over the current system and the first alternative. Architects were able to explain why the component was designed that way, using a non-technical speech. Management accepted the design for that components and asked to pass it to development team.

7. Future Research

First version and test of the Junior Architect Assistant needed a lot of manual work. Every form needed during the application case was created on paper or Word Processor. Also, the Pugh Selection Matrix template was made using Spreadsheet software. Every artifact was created and used separately.

A future research for an automated version, using software tool, will be made. There's the alternative to develop it from the beginning or to look for one and adapt it. At the moment, SEI is doing a similar effort with ArchE **[21][22]**, a tool running on the Eclipse platform. This tool is in Beta for now, and is described as a tool for aiding in the Decision Taking during Architecture Design. Current version supports decision making using a rule engine and reasoning frameworks; it also supports Quality Attributes Performance and Modifiability in the current version.

One of the most important aspects for the automated tool is related to feedback. Mechanisms must be defined to record reasoning after a decision is made, also, record the results of Pugh Concept Selection Matrix after evaluation. Fast access to this information should be a first priority for this functionality.

The 40 principles map will also be revised in order to map the principles to Quality Attributes Support. It was found after application case that it would be better to think in Quality Attributes for alternative generation and the principles should be a guidance for architects to achieve them.

8. Conclusion

First, we revised the work already done about TRIZ for Software. Most of the work has been done separately and no specific use or phase in the lifecycle was suggested or showed any application. Many authors affirmed TRIZ can be used for Software Development. Some claimed Architecture and High Level Design as the phases to be used, because it was too much for later phases. The 40 principles were the most discussed as being the most suitable tool for the study field. We also checked other TRIZ tools to find out which one can be used for software.

We described the method developed as an Assistance in the Architecture Design Process, introducing TRIZ for alternative generation, Pugh Selection Matrix for Alternative Evaluation and the description of how to use the Requirements Artifacts as method input.

Then, we tested the method in a small organization and studied the results. The process of Functional Analysis, Alternative Generation and Alternative Evaluation and Selection is explained in section 6.

TRIZ methodology helps us in finding a solution to a problem by applying existent and proven knowledge in different fields. We followed this to integrate a methodology with a series of existent and proven tools. Systematic Architecture Design and Evaluation was achieved by the integration of tools that solved one particular problem.

9. Bibliography and References

1. Katie Barry, Ellen Domb and Michael S. Slocum. What is Triz? Triz Journal.
2. Kevin C. Rea. Triz and Software – 40 Principle Analogies, Part 1 & 2. TRIZ Journal. September, 2001.
3. Kevin C. Rea. Applying TRIZ to software problems. Creatively bridging academia and practice in computing. Proceedings in TRIZCON2002. The Altshuller Institute Conference, May, 2002. The Triz Journal.
4. Darrell Mann. Triz for Software?. The Triz Journal, 2004
5. Darrell Mann. TRIZ And Software Innovation: Historical Perspective And An Application Case Study. TRIZCON2007. February, 2007.
6. Toru Nakagawa. Software Engineering and TRIZ. Structured programming reviewed with TRIZ. The TRIZ Journal. July 2005.
7. Bijay K. Jayaswal, Peter C. Patton. Design for Trustworthy Software: Tools, Techniques, and Methodology of Developing Robust Software, Chapter 12. Prentice Hall. August 31, 2006.
8. David Warburton. Getting better results in design concept selection. Medical Device Link. January 2004.
9. John Terninko, Alla Zusman, and Boris Zlotin. Systematic Innovation: Introduction to TRIZ. CRC. 1 edition. April 15, 1998.
10. Ron Fulbright. Triz and Software Fini. TRIZ Journal, 2004.
11. Kevin C. Rea. TRIZ for software. Using the Inventive Principles. Triz Journal. 2004.
12. Michael S. Slocum. Su-Field Analysis Model Solutions. TRIZ Journal.
13. Herman Hartmann, Ad Vermeulen and Martine van Beers. Application of TRIZ in Software Development. TRIZ Journal.
14. Alistair Cockburn. Use Cases. <http://alistair.cockburn.us> Revised: august 27th by Edgar Fernández.
15. Len Bass, Paul Clements, Rick Kazman. Software Architecture in Practice, Second Edition. Ch. 1. Addison Wesley, 2003.
16. Ellen Domb, H. William Dettmer. Breakthrough innovation in conflict resolution. Marrying TRIZ and the thinking process. TRIZ Journal.
17. Kalevi Rantanen, Ellen Domb. Simplified TRIZ. CRC Press LLC.
18. Philippe Kruchten. Architectural Blueprints—The “4+1” View Model of Software Architecture. IEEE Software. November, 1995.
19. Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, Judith Stafford. Documenting Software Architectures: Views and Beyond. Addison Wesley. 2002.
20. CMMI Product Team. CMMI® for Development, Version 1.2. CMU/SEI. August, 2006.

21. Felix Bachmann, Len Bass, Phil Bianco. Software Architecture Design with ArchE. CMU/SEI. 2007
22. Felix Bachmann, Len Bass, Mark Klein. Preliminary Design of ArchE: A Software Architecture Design Assistant. CMU/SEI 2003.
23. Mike Walmsley, Pugh Matrix: Decisions, Decisions, Decisions... Six Sigma First Group. 2005