Principios para la Formalización de la Ingeniería de Software

Sandro Javier Bolaños Castro [1]

Víctor Hugo Medina García [2]

Luis Joyanes Aguilar [3]

Resumen

En este artículo se presenta una propuesta para una ingeniería de software basada en la premisa de establecer un conjunto de principios necesarios para formalizar un cuerpo de conocimiento guiado no solo por actividades de trabajo, métodos y técnicas sino también fortalecida por propiedades que en efecto puedan ser utilizadas y aplicadas sin que haya discriminación en el tipo de proyecto que se realice.

Palabras clave: Ingeniería de Software, Proyecto, Principios, Formalización.

Principles for the Formalization of Software Engineering

Abstract

In this article a proposal is presented for a software engineering based on the premise of establishing a necessary set of principles to formalize a body of knowledge guided not alone for work activities, methods and technical but also strengthened by properties that indeed can be used and applied without there is discrimination in the project type that it is carried out.

Keywords: Software engineering, Project, Principles, Formalization.

1. INTRODUCCIÓN

Los procesos de software, que se constituyen como una de las principales herramientas para hacer ingeniería de software, deben plantear el desarrollo como un proceso guiado no solo por un conjunto de actividades de proceso, sino que además deben apoyarse en un cuerpo teórico de principios y leyes que permitan articular las tareas de manera formal y controlada. Este planteamiento releva el individuo sin descuidar la organización y la naturaleza singular del producto. La responsabilidad de los flujos de trabajo y actividades no solo debe estar en suministrar un método de desarrollo, aún más importante es establecer criterios formales del manejo de la complejidad, incertidumbre, corroboración, singularidad y la rastreabilidad del conocimiento que constituye el insumo fundamental en el proceso.

La propuesta que se presenta pretende impactar las formas convencionales de desarrollar software y con ello replantear los enfoques clásicos que en cuanto a procesos se hacen en ingeniería de software. Esto traería consecuencias favorables como la disminución de fracasos en desarrollo, el manejo del software como conocimiento, fortalecimiento del cuerpo de conocimiento de la ingeniería de software, tratamiento del problema a diferentes niveles de la organización, integración de disciplinas, vista holística de los problemas de desarrollo, desarrollo de productos de calidad, entre otras.

2. EL MÉTODO COMO INSTRUMENTO FUNDAMENTAL EN INGENIERÍA DE SOFTWARE

Las metodologías y procesos de desarrollo de software, han suplido por un espacio de tiempo importante las necesidades de abordar problemas de desarrollo de productos informáticos, sin embargo existe gran evidencia, que no han sido suficientes para dar soluciones satisfactorias a la gran diversidad y complejidad de los retos que se plantean en la disciplina. Existe una lucha fuerte de propuestas que pasan desde el formalismo y dogmatismo como los casos de CMMI, PSP y TSP, entre otros; hasta las propuestas ágiles y pragmáticas como Scrum [1], XP [2], Cristal [3], Evo [6], ASD [7], DRA [13], entre otros. Cada estilo de proceso o metodología reclama para si la supremacía y la verdad en cuanto a desarrollo se refiere. Lo cierto es que no hay un santo grial de proceso y parecería incluso que el problema no esta en las actividades que se desarrollen sino en un faltante que no se encuentra en el proceso en si, quizá cuando Karl R. Popper afirma: *"Por regla general, empiezo mis clases sobre el método científico diciendo a mis alumnos que el método científico no existe… Solo hay problemas y el impulso de resolverlos…"* [12]. Se nos esta planteando un importante cuestionamiento en la misma utilización de los procesos y metodologías.

Sin bien es cierto, los procesos y más específicamente las actividades de proceso configuran un cuerpo disciplinar fundamental para resolver los problemas en esta área, también es cierto que esta disciplina en particular adolece de un componente científico de facto que le permita explorar de manera formal y con experimentación directa los fenómenos que se presentan en ella. Mientras el método científico en la mayoría disciplinas es de uso directo en ingeniera de software no es tan evidente, al fin de al cabo el

[1 y 2] Profesores Facultad de Ingeniería de la Universidad Distrital Francisco José de Caldas. Bogotá, Colombia. Miembros del grupo de Investigación GICOGE.

[3] Profesor Facultad de Informática de la Universidad Pontificia de Salamanca campus de Madrid - España. Director del grupo de investigación: Ingeniería del Software y Sociedad de la Información y el Conocimiento, Universidad Pontificia de Salamanca.

Artículo recibido en Junio de 2008, aprobado para publicación en Agosto de 2008

método científico permite seguir una rutina basada en fenómenos evidenciables y perceptibles de forma natural, en tanto que la disciplina del software es eminentemente artificial.

3. PRINCIPIOS PARA INGENIERÍA DE SOFTWARE

Los procesos de desarrollo de software definitivamente no se pueden dejar de lado, en lugar de ello se deben fortalecer, es decir se debe armar un cuerpo disciplinar soportado en conjunto de principios, que permitan ajustar los problemas de manera formal, sin ambigüedades, dejando de lado la especulación y las prácticas ad hoc.

Los defectos más comunes de los procesos están en considerar una serie de prácticas que funcionaron en determinadas situaciones. Pero no pasan de ser prácticas, que tarde o temprano no tendrán sentido en escenarios diferentes. Las tendencias sobre todo de los métodos heterodoxos están muy ligadas a la importancia que se da a las personas, el mismo manifiesto ágil propone: "Los individuos y la interacción por encima de los procesos y herramientas". Y no es que las personas no sean importantes en la búsqueda del conocimiento, lo que es un punto débil es que el método de llegar al conocimiento dependa de la persona, la disciplina por si sola debe brindar los mecanismos formales que permitan seguir la senda al conocimiento, en este caso la senda al desarrollo. En otros palabras *los fenómenos suceden para su naturaleza intrínseca y no por al interpretación de subjetiva de las personas que lo experimentan*.

Apuntando en la dirección de dar una directriz a la disciplina de software se formulan cinco principios fundamentales que restringen de manera formal el que hacer en esta área del conocimiento, así:

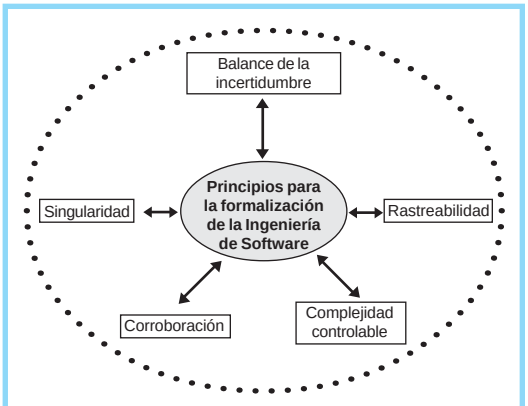

Figura 1. Principios para la formalización de la Ingeniería de Software. Fuente: Los autores.

3.1 Balance de la Incertidumbre

El desarrollo de software es un ejercicio creativo en el que se parte de un dominio del problema con gran incertidumbre para llegar a un dominio de la solución con gran certidumbre. En el dominio del problema hay más preguntas que respuestas, mientras en el dominio de la solución hay más respuesta que preguntas, por lo menos con respecto al problema resuelto. En este orden de ideas es de suponer que el ejercicio de desarrollar software esta en ir eliminando paulatinamente la incertidumbre para acercarse a la certidumbre permitida que nos de la solución. En este camino es de esperarse que se busquen mecanismos que faciliten el entendimiento del problema.

Aparecen dos constantes importantes *las preguntas y las repuestas*. Por un lado las preguntas se originan de las posibilidades que puede tomar un problema mientras las respuestas se originan de las soluciones que se pueden adoptar ante un problema. Las preguntas son necesarias y pertinentes para resolver un problema, de no ser así tendríamos un problema predecible en donde no hay mas posibilidades que la obtenida en un primer razonamiento.

Por ejemplo, un *algoritmo* generalmente esta dotado de preguntas pues es implícito en el las posibilidades o decisiones que se pueden adoptar ante un problema. El problema no radica en eliminar las preguntas, aunque sería bastante provechoso, sino más bien ante la inminencia de estas, realizar las mejores preguntas y que apunten a los orígenes del problema, cuyas soluciones necesariamente redunden en la solución del problema. Como premisa fundamental de las preguntas se podría plantear:

- *Todo problema por la inminencia de la incertidumbre produce preguntas.*
- *Si hay ausencia de preguntas por la naturaleza particular del problema, se tiene entonces un estado de certeza.*

Por su parte las respuestas son el resultado de las preguntas o de los estados de certeza de un problema. Estas deben ofrecer la solución a un problema. Desde otro punto de vista Se podría plantear que un problema se evidencia a través de las preguntas; mientras la solución se evidencia a través de las respuestas. Bajo esta consideración es de esperarse entonces que necesariamente, o por lo menos en su gran mayoría se produzca la dupla *pregunta-respuesta*. Desde un planteamiento deductivo podría pensarse en un marco de trabajo en donde no se vean problemas sino simplemente soluciones por tanto se eliminaría el complejo mundo de las preguntas, sin embargo este mecanismo es mucho mas complejo aunque no lejano del software.

Las respuestas están en el lado de la solución pero son originadas por preguntas o son en esencia un estado de verdad transitorio para un problema. En este orden de ideas se podrían plantear en la siguiente premisa:

- *Toda respuesta o proviene de una pregunta o de un estado transitorio de verdad.*

32

El desarrollo de un problema basado en un esquema lógico de programación sugiere que se realicen preguntas y respuestas, las cuales posibilitan que la transición de estados contenga la lógica necesaria para mostrar una solución. Ante estas evidencias vale la pena cuestionarse sobre la correlación que deben tener las preguntas y las repuestas, y producto de esta correlación podrían originarse los siguientes planteamientos:

1) ¿Una pregunta debería originar una sola respuesta?
2) ¿Una pregunta debería originar varias respuestas?
3) ¿Varias preguntas deberían originar una sola respuesta?

Si se evalúa el segundo planteamiento se podría pensar en que el problema expresado en una única pregunta es resuelto a través de varios repuestas de las cuales alguna de ellas satisfacería la pregunta o simplemente de manera oculta de lo que se dispone como aparentes respuesta es de un protocolo en el que todas las aparentes repuestas hacen parte de un misma solución. En el primer caso el problema puede ser resulto con una de las respuestas en el segundo con la suma de todas las respuesta.

En el tercer planteamiento se estaría ante una situación en la que se podría pensar en una respuesta inteligente capaz de proveer la solución a varios cuestionamientos, o simplemente se estaría ante la evidencia de la redundancia de preguntas que conduce a la misma repuesta. En el primer caso en el que la respuesta es una conclusión inteligente que resuelve varios cuestionamientos se esta ante un abstracción fuerte del problema en el que se debería descartar la redundancia, es de manejo cuidadoso pues las posibles combinaciones lógicas de las preguntas pueden apuntar a soluciones diversas que no se han visto con facilidad, cabe preguntarse además del espectro de las preguntas que se solucionan con una única respuestas, cual es su complemento pues en el puede haber elementos ocultos de la solución del problema. Por otro lado puede existir redundancia de preguntas y ante lo que parecen varios cuestionamientos puede resultar más bien malos planteamientos.

El segundo y tercer planteamiento apuntan a un reflexión importante, ¿están en la muchas preguntas o muchas respuestas reflejado verdaderamente un problema de redundancia que debería ser normalizado para logran un balance en el que las preguntas podrían corresponder uno a uno con las respuestas? Y no es porque se trate de eliminar la riqueza de la pregunta o de la respuesta; más bien de lo que se trata es de converger los cuestionamientos y las soluciones. Esto trae ventajas en el manejo de la situación problemica. Pensemos en algunos ejemplos:

1) a.¿Qué número resulta de 10/2?
   b. ¿Cuál es el numero de dedos de una mano?

Las anteriores preguntas motivan una respuesta el número 5. Esta redundancia de preguntas podría centralizarse en una sola preguntas similar a: ¿Cual es la mitad de los dedos de la mano? dentro de la cual se expresa las preguntas a y b.

2) ¿Cuál es el conjunto de 5 números naturales? respuesta que origina un conjunto infinito. Podría normalizarse a al conjunto {1, 2, 3, 4, 5}.

En las dos consideraciones anteriores se puede ver el concepto pregunta respuesta, además como se puede generar la normalización de la repuesta o la pregunta; esta normalización que produce asignación uno a uno de preguntas y respuestas es lo que se denomina equilibrio del balance de la incertidumbre. Una buena forma de seguir el progreso del desarrollo de software es la de estimar el balance este puede ser de tres formas, tal y como se ilustra en la Fig. 2.

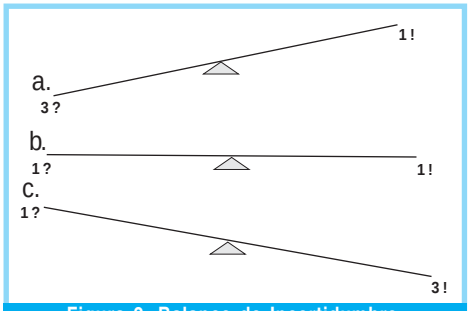

Figura 2. Balance de Incertidumbre.
Fuente: Los autores.

El primer tipo de balance, aquel en el cual pesan más las preguntas que las repuestas, este tipo de balance es de alta incertidumbre, existen más preguntas que respuesta y puede desencadenar difícil manejo en los desarrollos. El tipo de balance en donde las preguntas coinciden con las repuestas es el tipo de balance equilibrado el cual es el más aconsejado como medio de atacar la incertidumbre, pues cada interrogante tiene su solución. El tercer tipo de balance en el que es de mayor peso las repuestas que las preguntas, generalmente obedece a un conjunto de respuestas que hacen parte de protocolos o de respuestas múltiples. El mejor balance es el equilibrado pues se tiene una correspondencia uno a uno del problema y la solución. Para lograr balancear equilibradamente el software es necesario eliminar la redundancia en las preguntas o en las respuestas con nuevas preguntas o nuevas respuestas mejor establecidas.

3.2 Singularidad

El debate sobre las metodologías y procesos de desarrollo puede tener tanto de ancho como de profundo, quizá el problema en si no lo constituya el proceso sino el producto. Querer desarrollar un producto de software por medio de un proceso atado a ideas convencionales puede ser la fuente del problema. El software es un producto con propiedades particulares (ver Fig.3).

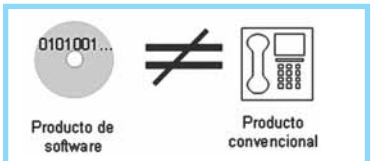

Figura 3. Software como producto.
Fuente: Los autores.

- No es perceptible a los sentidos: es decir no se puede oler, degustar, tocar ver u oír, es una representación que existe en otros medios de almacenamiento y reproducción.
- Vive en el mundo binario y no se ajusta en su naturaleza misma a leyes fácticas.
- Es replicable fácilmente por tanto es fácil de reproducir.
- No se degrada aunque si se pueda desactualizar.
- Puede ser integrado y crear complejidades mayores.
- Es consecuencia de la lógica y el raciocinio.

Al ser el software imperceptible a los sentidos se pierde el poder de experimentación que podría realizarse directamente sobre él, por lo menos la experimentación convencional apoyada en las ciencias fácticas. El software entra mas en la categoría de un ente formal producto de nuestro pensamiento y por ende obedece a él. Como su nicho es la representación binaria no se puede controlar con un ciencia fáctica por tanto sigue mas en el terreno de lo formal. Se puede replicar fácilmente por tanto su producción es una sola vez. Como no se degrada, su ciclo de vida depende de la vigencia que pueda tener. Al ser integrado para crear complejidades mayores es escalable. Como es consecuencia de la lógica y el raciocinio depende del desarrollo paradigmático con el que se cuente.

Estas características pueden ser benéficas o no si se compara con un producto convencional. Por ejemplo un producto que se puede manipular con los sentidos es predecible y manejable a través de disciplinas fácticas como la biología, química, y la física, sobre el software no se puede hacer un experimento directo que se ajuste a la ciencias básicas por tanto se pierde la fortaleza que esa experimentación puede dar, se debe entonces formular un ente formal que sea la representación del software y sobre el deducir comportamientos y leyes. Por su puesto esto es una tarea que reúne una gran complejidad. El que el software viva en el mundo binario tiene sus pros y contras, el espacio de problema por lo menos se sabe es restringido a este dominio pero quizá lo poco o mucho que hay al respecto parece no ser suficiente para controlarlo. El que sobre el software se haga replica con facilidad es una ventaja con respecto a un producto convencional, mientras el fabricar un producto convencional requiere de un complejo logístico que se debe utilizar una y otra vez; con el software no sucede, tan solo es necesario que se produzca una vez, pues después se replica con medios de reproducción de la lógica implantada. Al no degradarse el software no se le puede hacer un conteo de depreciación como sucede con un producto convencional esto es una gran ventaja pues este efecto negativo se puede evitar, el concepto mas parecido es el de desactualización que por su puesto sucede con cualquier otro producto. El que posibilite la integración para crear complejidades mayores permite un doble uso del software; uno como la unidad funcional para la que fue creado y otro como un unidad componente de un sistema mayor, propiedad poco común en otros productos. Al ser este, producto del raciocinio y la lógica también es posible pensarlo como el producto de la creatividad y la innovación características que lo hacen relevante frente a otros productos.

Infortunadamente el software no se puede oler, ver, tocar o saborear convencionalmente pero si se le pueden asociar propiedades similares, esto a través de sus modelos en diseño e implementación, no es fácil pensar en medir un programa de hecho la misma medida resulta ser bastante relativa, aunque se puede asociarle al software el peso, como el número de bytes que contenga, lo que generalmente se conoce como el tamaño, este no necesariamente es un índice de calidad, tan solo es un índice de almacenamiento. La física, la biología y la química como disciplinas fácticas no pueden ser asociadas al software como convencionalmente se hace con otras disciplinas de ingeniería, lo más cercano son las ciencias formales: matemáticas, y sobre todo la lógica (ver Fig. 4).

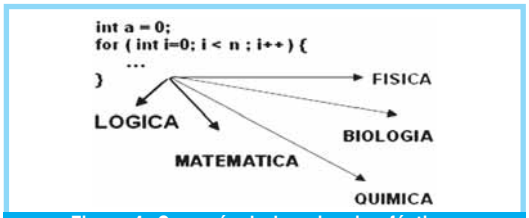

Figura 4. Cercanía de las ciencias fácticas y formales. Fuente: Los autores.

3.3 Complejidad Controlable

Desarrollar software implica un grado de complejidad considerable que se encuentra presente en todas y cada una de las actividades de proceso que se realicen para alcanzar este objetivo. Esta complejidad puede ser vista así:

$$\mathbf{CTP \geq \Sigma\ CAPi}$$

La complejidad del proceso CTP, es mayor o por lo menos igual a la sumatoria de las complejidades de las actividades de proceso i, CPAi. Es de esperar este razonamiento si se tiene en cuenta que la suma de las partes es mayor u igual al todo.

Teniendo en cuenta todas y cada una de las actividades principales del proceso la complejidad puede ser vista como:

$$\Sigma\ CAPi = C_{ingdereq} + C_{diseño} + C_{implementacion} + C_{prueba} + C_{mantenimiento}$$

Las complejidades de requerimientos, diseño, implementación, prueba y mantenimiento se pueden ver como:

$$C_{ingdereq} \geq C_{comunicacion} + C_{representacion}$$

$$C_{diseño} \geq C_{arquitectura} + C_{diseñodebajonivel}$$

$$C_{implementacion} \geq C_{lenguaje} + C_{paradigma} + C_{algoritmo}$$

$$C_{prueba} \geq C_{elabDeExperimentos} + C_{aplicDeExperimentos}$$

$$C_{mantenimiento} \geq C_{estabilizacion} + C_{mejoramiento}$$

Los retos fundamentales de la complejidad de realizar la ingeniería de requerimientos están centrados en la comunicación y la representación. Por un lado la comunicación implica un proceso complejo de interacción de seres humanos y sistemas. No se pueden asumir que la obtención de requerimientos se reduce a la aplicación de métodos convencionales como: encuestas, entrevistas, cuestionarios, entre otras técnicas comunes. Los intereses y demás fenómenos que se producen por la relación que entablan las personas en la constitución de sistemas son de una índole amplia por involucrar personas y sistemas artificiales; que se han resumido en una vista parcializada desde la perspectiva de la ingeniería, sin abordar el problema desde la dimensión humana que resulta siendo en últimas la mas importante. Los sistemas que se desarrollen son para y por las personas, en esta dimensión un problema debería abordarse en un amplio espectro que involucre una elaborada sinergia de las disciplinas humanas y de ingeniería. Cuando se ha resuelto el problema de comunicación es necesario resolver el otro problema fundamental, el cual se encuentra en la necesidad de la representación. A pesar de tener frameworks y métodos de análisis y modelado de requerimientos, como por ejemplo los casos de uso [8], las historias de usuario [2], entre otras. Seria optimista pensar que estas herramientas son suficientes para resolver el problema de representación. La representación exige que sea fiel a lo representado y que a su vez sea simple. Sumar simultáneamente estas dos características no es sencillo, si se tiene en cuenta que un modelo fiel a lo planteado involucra complejidad y por el contrario un modelo simple termina siendo incompleto. Parece que las dos propiedades necesarias de representación entran en conflicto. Queda aquí un reto importante de revisar, que una vez resuelto puede llevarnos a un gran avance en ingeniería de requerimientos.

Por su lado el diseño tiene una complejidad en la arquitectura y el detalle. La arquitectura como disciplina emergente tiene sus propios retos, pero quizá se deba prestar atención a la necesidad de contar con modelos estandarizados que sirvan como mapas de navegación, al respecto hay importantes aportes, Shaw y Garlan [14] apunta a una clasificación de alto nivel de posibles modelos arquitectónicos, el trabajo de la pandilla de los cuatro GoF [5], es otra importante aproximación al problema. Los lenguajes MIL y ADL[1] también son un importante esfuerzo al respecto incluso UML 2.0 [8] se ha preocupado por dar soporte al problema Últimamente MDA [9] y el estándar MOF[2]. Afortunadamente se es conciente de la necesidad de aproximar el modelo expresado en lenguajes arquitectónicos a los modelos expresados en lenguajes de programación. Gran parte de la confianza en hacer arquitectura se centra precisamente reducir la distancia que tiene el modelo del diseño y el modelo de la implementación. Como en la práctica los modelos de la implementación son los funcionales, le reducen gran impacto a los modelos arquitectónicos. Podría plantearse entonces el por qué mejor no evitarlos, si terminan distando de los modelo que realmente funcionan. Infortunadamente los modelos de implementación requieren detalles en los cuales es peligroso entrar en fases tempranas del desarrollo.

La complejidad de la implementación tiene su fuente en los leguajes de programación los paradigmas y los algoritmos. Si todo fuera tan sencillo como antes de la torre de babel, sin embargo la diversidad de lenguajes de programación y plataformas de desarrollo al contrario de hacerle un bien a la disciplina, produce problemas como la portabilidad la incompatibilidad entre otras. Un lenguaje es complejo desde la misma sintaxis, pasando por los mecanismos de gestión de memoria hasta la interpretación de los paradigmas. Los paradigmas de desarrollo similarmente a lo planteado por Tomas Kuhn [10], recurre a un conjunto de teorías, métodos y prácticas para la representación de un problema en términos de un lenguaje de alto nivel. A pesar de contar con nivel importante de abstracción los lenguajes aun distan de parecerse al lenguaje natural, el camino recorrido desde la expresión del sistema computacional al lenguaje natural, aun se encuentra corto. De ahí la necesidad de otros mecanismos y niveles de abstracción. Finalmente los algoritmos introducen su propia complejidad, al respecto hay importantes estudios sin embargo esto generalmente apuntan en una dirección, el rendimiento que pueden producir, abandonando otras perspectivas como la facilidad de mantenimiento, escalamiento, portabilidad entre otras propiedades.

La prueba es una actividad de proceso que consume gran parte de los recursos del desarrollo, incluso invirtiendo grandes recursos es imposible realizar todo el conjunto de pruebas posible [11]. Hay un importante trabajo al respecto, pero gran parte del esfuerzo termina siendo labor personal, evidencia de que la disciplina de pruebas se debe formalizar no solo en su aplicación sino que además debe fortalecer su cuerpo de conocimiento. Lo que debería fortalecerse para mejorar esta actividad de proceso, es la planificación disciplinada del diseño de experimentos, teniendo en cuenta que la prueba se puede establecer como un experimento. Los

[1] Lenguajes de interconexión modular y lenguajes de descripción arquitectónica [13].

[2] MOF meta-object facility, http://www.omg.org/technology/documents/formal/mof.htm

experimentaos configuran un conjunto de pruebas con el objetivo de generar datos, que al ser analizados estadísticamente, proporcionan evidencias objetivas que permitan resolver los interrogantes planteados por el experimentador, sobre determinada situación [4]. El diseño de experimentos permite la introducción de la estadística con la cual es posible formalizar la prueba, el uso de un estadístico genera un nivel de confianza elevado a una disciplina que aun tiene muchos elementos de subjetividad. Una vez se haya diseñado el experimento es necesario aplicarlo, esto puede ser tan complejo como el mismo diseño, pero es en últimas la aplicación la que resulta trascendente, pues es con ella que se pone a prueba las hipótesis que se formulan. Las pruebas de software tienen un gran ingrediente científico en el diseño de experimentos, se debe aprovechar esta fortaleza para estabilizar una disciplina que resulta siendo más compleja en muchas ocasiones que el mismo desarrollo.

Por otro lado el mantenimiento introduce dos tramos de complejidad, la complejidad ocasionada por el mejoramiento que el software sufre, y la estabilización producto de los posibles cambios que se den con el mejoramiento. Al software le es inherente el mejoramiento continuo, característica que es propuesta en varios modelos de proceso, incluso es la principal característica de propuestas como la de CMMI. El mejorar el software es equivalente al mejoramiento progresivo de los modelo propios del software sean estos de análisis, diseño o implementación. Las dimensiones del mantenimiento deben verse de manera más amplia a lo que clásicamente se trata, y debe ayudar al perfeccionamiento y la fiabilidad misma de ese perfeccionamiento.

3.4 Corroboración

La corroboración, es la propiedad asociada a la capacidad de aprobación de los diferentes modelos del proceso de desarrollo. La corroboración es un concepto ampliado de prueba, necesario precisamente para obtener confianza en los modelos que se realizan a lo largo del proceso de desarrollo. Gran parte de la problemática ocasionada por la poca fiabilidad de una implementación es producida por la sobrecarga que tiene el modelo de implementación "código en un lenguaje de programación", esta carga es la carga de defectos altamente probables de los modelos anteriores a la implementación y que no so corregidos al momento de implementar.

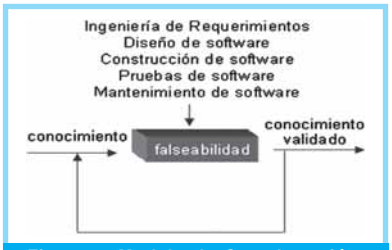

Figura 5. Modelo de Corroboración. Fuente: Los autores.

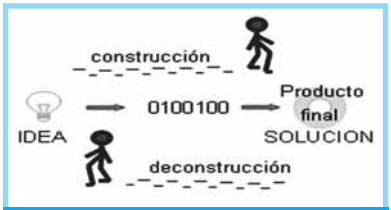

Figura 6. Rastreabilidad. Fuente: Los autores.

Un modelo de análisis arroja errores de análisis, un modelo de diseño y arquitectura arroja errores propios del diseño y arquitectura, estos errores no se perciben hasta que se plantea el modelo de implementación que es sobre el que concretamente se puede probar. El efecto colateral que produce la corroboración es la fuerte necesidad de aprobar cada uno de los artefactos del desarrollo. Como evidentemente esta tarea no es tan común en modelos de análisis y diseño, parece extraña incluso imposible. Pero gran parte del esfuerzo podría centrase en logara tener lenguajes para cada uno de esto modelos, remplazando las técnicas que son poco formales. Los lenguajes son mas predecibles y parametrizados, por su puesto no es tarea fácil, pero indudablemente es una necesidad. Incluso el que se desarrollen lenguajes para otros modelos diferentes a los de implementación, posibilitaría abordar el problema de trazabilidad y alejarnos más de los detalles engorrosos de los lenguajes de programación. Un modelo de corroboración puede verse como en la Fig. 5.

En el modelo de corroboración se identifican dos entradas por un lado el conocimiento y por otro lado las actividades de proceso: ingeniería de requerimientos, diseño de software, construcción de software, pruebas de software y mantenimiento de software. El procesamiento de estas entradas en una maquina de falseabilidad, arroja una salida estimada como el conocimiento validado. La primera entrada referida al conocimiento se constituye como el conjunto de técnicas, métodos y prácticas necesarias para la corroboración, por otro lado cada una de las actividades de proceso produce artefactos, los que a su vez deben ser seguibles a través de modelos y ojala lenguajes propios. La combinación del conocimiento necesario pera la actividades de procesos entran en una maquina de falseación, cuyo objetivo fundamental es poner a prueba las hipótesis que se formulen de la entradas, tratando de demostrar que no es correcta, después del procesos de falseación se produce una clase de conocimiento que se considera validado, ya sea que en este sea pudo falsear o acertar. La misma prueba puede ser falseada, sentido en el cual la corroboración puede ser considerada incluso como una metaprueba.

3.5 Rastreabilidad

La rastreabilidad es otra propiedad que se debe conseguir en un software, esta podría confundirse con la trazabilidad pero al igual de la corroboración y prueba, la rastreabilidad y trazabilidad tienen una correspondencia similar. La rastreabilidad involucra un camino doble de construcción y de construcción de un concepto. Mientras en la construcción de un concepto se involucran un conocimiento ordenado y disciplinado ascendente, en la deconstrucción se involucra una revisión del conocimiento con la

posibilidad de seguir ese conocimiento de manera no solo sencilla sino constructiva de nuevo conocimiento (ver Fig. 6).

Los procesos de la rastreabilidad, construcción y deconstrucción permiten la elaboración de conocimiento en ambas direcciones, ya sea de la ida hacia el producto o tomando el mismo producto hasta reconstruir la idea que lo origino. En estos dos caminos es posible no solo hacer seguimiento sino construcción de conocimiento. La rastreabilidad permite fortalecer disciplinas como la ingeniería inversa. El proceso de rastreabilidad visto desde otra perspectiva facilita la sinergia de las actividades de proceso las cuales no necesariamente deben ser lineales. Similarmente a seguir migas de pan, para adentrarse al bosque y poder salir sin perderse "Hansel y Gretel" la rastreabilidad permite sumergirse en el desarrollo construyendo conocimiento y regresar deconstruyendo, con la ganancia adicional de evolucionar en la concepción del producto.

CONCLUSIONES

La ingeniería de software como área disciplinar, aún tiene muchos vacíos en el cual se pueden hacer importantes aportaciones, evidencia de esos vacíos, es la estadística poco disminuida de fracasos que aún se tiene.

Los problemas en ingeniería de software están estrechamente asociados a la falta de un cuerpo de conocimiento más amplio. Es optimista creer que tan solo con el uso de los procesos se puedan obtener productos de calidad, es necesario además constituir un conjunto de principios que formalicen el desarrollo mismo y deje de ser una disciplina que descargue gran parte de la responsabilidad en las técnicas y habilidades personales.

Los principios aquí expuestos quieren poner de manifiesto la necesidad de utilizar un formalismo que aborde la problemática de manera no solo estandarizada sino y aun mas importante de manera objetiva.

En la contrastación de la ingeniería de software con otras disciplinas, si bien es cierto se tienen grandes similitudes en la mayoría de los caso estas son aparentes, y mas bien se reclama un cuerpo de conocimiento propio dotado de principios leyes y axiomas propios de esta área que es mas de corte artificial que de corte natural.

REFERENCIAS BIBLIOGRÁFICAS

[1] Beedle Mike, Devos Martine, Sharon Yonat, Schwaber Ken and Sutherland Jeff. "SCRUM: A pattern language for hyperproductive software development". En N. Harrison, B. 2. Foote, and H. Rohnert, eds., Pattern Languages of Program Design, vol. 4, pp. 637-651. Reading, Addison-Wesley. 1998.

[2] Beck Kent. Extreme Programming Explained: Embrace Change. Reading, Addison Wesley. 1999.

[3] Cockburn Alistair. "Crystal Clear. A human-powered methodology for small teams, including The Seven Properties of Effective Software Projects". Borrador. Humans and Technology, versión del 27 de febrero 2002.

[4] Gutiérrez Pulido Humberto y De la Vara Salazar Román. "Análisis y Diseño de Experimentos". Quinta edición. Ed. Mc Graw Hill. 2008.

[5] Gamma Erich, Helm Richard, Jonson Raplh and Vlissides Jhon. "Design Patterns, Elements of reusable object-oriented software". Ed. Addsion Wesley. 1995.

[6] Gilb Kai. Evolutionary Project Management & Product Development (or The Whirlwind Manuscript), http://www.gilb.com/Download/EvoProjectMan.pdf. 2003.

[7] Highsmith Jim. Adaptive software development: A collaborative approach to managing complex systems. Nueva York, Dorset House. 2000.

[8] Jacobson Ivar, Booch Grady and Rumbaugh James. "The Unified Modeling Language Referente Manual" Second edition. Ed Addison Wesley. 2005.

[9] Kleppe Anneke, Warmer Jos and Basts Wim. "MDA Explained, The Model Driven Architecture: Practice and Promise". Ed. Pearson Education. 2003.

[10] Kuhn Thomas S., "La Estructura de las revoluciones cientificas". Trad, Ed.Fondo de Cultura Econocmica. 2006.

[11] Patton Ron. "Software Testing". Ed. Sams. 2006.

[12] Popper Kart R.. "Realismo y el objetivo de la ciencia", Segunda edición. Ed. Tecnos, S.A, 1998.

[13] Pressman Roger. "Ingeniería del Software", Sexta edición. Ed. Mc Graw Hill. 2005.

[14] Shaw Mary and Garlan David. "Software Architecture, Perspectives on an Emerging Discipline". Ed. Prentice-Hall. 1996.

Sandro Javier Bolaños Castro

DEA en el Doctorado en Ingeniería Informática de la Universidad Pontifica de Salamanca, campus de Madrid. Magíster en Teleinformática de la Universidad Distrital Francisco José de Caldas. Ingeniero de Sistemas de la Universidad Distrital Francisco José de Caldas. Actualmente es profesor asociado en la Universidad Distrital, en la Maestría en Ciencias de la Información y las Comunicaciones, en la Especialización en Ingeniería de software y en el Programa Curricular de Ingeniería de Sistemas. Área de Ingeniería de Software. Pertenece al grupo de investigación en comunicaciones, informática y gestión del conocimiento - GICOGE. **sbolanos@udistrital.edu.co**

Víctor Hugo Medina García

PhD. Ingeniería Informática de la Universidad Pontifica de Salamanca campus de Madrid. Magíster en Informática de la Universidad Politécnica de Madrid. Especialista en Marketing de la Universidad del Rosario. Ingeniero de Sistemas de la Universidad Distrital Francisco José de Caldas. Actualmente es profesor titular en la Universidad Distrital, en la Maestría en Ciencias de la Información y las Comunicaciones y el Programa Curricular de Ingeniería de Sistemas, en el área de Ingeniería de Software y Gestión del Conocimiento. Es investigador del grupo GICOGE. Profesor asociado de la Universidad Pontifica de Salamanca y profesor invitado en la Universidad de Oviedo. **vmedina@udistrital.edu.co**

Luis Joyanes Aguilar

Doctor en Informática de la Universidad de Oviedo. Doctor en Ciencias Políticas y Sociología de la Universidad Pontifica de Salamanca campus de Madrid. Licenciado en Ciencias Físicas de la Universidad Complutense de Madrid. Licenciado en Enseñanza Superior Militar de la Academia Militar de Zaragoza. Profesor Titular Agregado de Cátedra de Lenguajes y Sistemas Informáticos de la Facultad de Informática de la Universidad Pontificia de Salamanca. Se desempeñó como Decano de la Facultad de Informática, Director de Doctorado, Director de Masters y Director del Dpto de Lenguajes, Sistemas Informáticos e Ingeniería de Software, en la Universidad Pontifica de Salamanca campus de Madrid. Autor de mas de 40 libros de Informática y numerosos artículos en revistas y congresos internacionales.