No Silver bullets

Todo desarrollo de software comprende tareas tanto “esenciales” como “accidentales” referentes al desarrollo. La mayor parte del trabajo de ingeniería está dedicado a resolver los problemas “accidentales” del software, como lo son barreras de hardware o lenguajes de programación engorrosos, pero también es necesario abordar los problemas “esenciales” del software para lograr un avance significativo.

Los proyectos de software suelen parecer simples, sin complicaciones, pero tienen la capacidad de convertirse en “monstruos” con grandes retrasos, presupuestos inflados y productos fallidos. Por lo que se suele buscar desesperadamente una “bala de plata”, algo que haga que los costos del desarrollo de software bajen tan rápidamente como lo hacen los costos del hardware.

Pero no existe una “bala de plata”, no existe un único avance ya sea tecnológico o en técnicas de gestión que sea capaz de producir un una mejora significativa en la productividad, fiabilidad o simplicidad por si mismo.

Si bien, no existe una única solución que pueda brindar avances significativos por si misma, van apareciendo varias innovaciones prometedoras, que comprometiéndose a mejorarlas, propagarlas y explotarlas, con el tiempo pueden terminar con un avance significativo ya sea en productividad, simplicidad o fiabilidad.

No es solo que no hay una bala de plata a la vista, sino que la misma naturaleza del software hace que sea improbable que aparezca alguna.

Para entender mejor las dificultades que presenta el software, se las puede dividir en dos. Por un lado está la “esencia” del software, que son aquellas dificultades inherentes a la naturaleza del software, por el otro están los “accidentes”, que son aquellas dificultades que aparecen en durante el desarrollo de software pero que no están en su naturaleza, como las complejidades de los lenguajes de programación.

La esencia del software es una construcción de conceptos interconectados: conjuntos de datos, relaciones entre los datos, algoritmos e invocaciones de funciones. Esta esencia nos indica que uno de estos conceptos abstractos construidos tiene muchas representaciones. Sin embargo es muy preciso y muy detallado.

La parte difícil de construir software es la especificación, diseño y pruebas de el concepto, no el trabajo de representarlo y probar la fidelidad de esta representación. Si se cometen errores de sintaxis, pero son triviales comparados con los errores conceptuales en muchos sistemas.

Si esto es cierto, construir software va a ser siempre difícil, inherentemente no existe una bala de plata.

Las propiedades inherentes a la esencia del software moderno son: complejidad, conformidad, adaptabilidad, e invisibilidad.

**Complejidad:** Las entidades de software con más complejas por su tamaño, que quizás cualquier otra construcción realizada por los humanos, debido a que no existen dos partes iguales. Si existen similaridades, se juntan en una subrutina. En este sentido, el software difiere de vehículos, edificios o computadoras, en donde los elementos repetidos abundan.

Las computadoras digitales son más complejas que la mayoría de las cosas que ha construido la humanidad, tienen una gran cantidad de estados, lo que hace que sea difícil encontrarlos, describirlos y testearlos. El software tiene muchísimos más estados que una computadora.

Debido a que escalar una entidad de software no es simplemente una repetición de los mismos elementos a mayor escala, es necesario un incremento en el número de diferentes elementos.

En la mayoría de los casos, los elementos interactúan entre ellos de una forma no lineal, y la complejidad del todo incrementa mucho más que linealmente.

Esta complejidad del software es una propiedad esencial, no un accidente. Por siglos los matemáticos y físicos han realizado grandes avances al construir modelos simplificados de fenómenos complejos. Esto funciona porque las complejidades ignoradas en los modelos no son propiedades esenciales del fenómeno, esto no funciona cuando la complejidad es parte de la esencia del problema.

Muchos de los problemas clásicos de desarrollar software derivan de esta complejidad esencial, que no incrementa linealmente respecto a su tamaño. A partir de esta complejidad surgen problemas de comunicación entre miembros del equipo, lo que lleva a fallos en el producto, costos excesivos, retrasos. De esta complejidad viene la dificultad de enumerar, y mucho menos entender, todos los posibles estados que puede tener el software, y de ahí surge la falta de fiabilidad. De la complejidad de la estructura surge la dificultad de extender los programas con nuevas funcionalidades sin crear efectos secundarios, también de allí surgen los estados no identificados, que pueden llevar a problemas de seguridad.

No solo los problemas técnicos surgen de la complejidad, sino también los problemas administrativos. Esto hace que tener una vista general sea difícil, impidiendo una integridad conceptual, y haciendo que sea difícil encontrar y controlar todos los cabos sueltos. Esto lleva a una gran carga de aprendizaje y entendimiento del personal, lo que puede llevar al desastre.

**Conformidad:** Gran parte de la complejidad que debe atender un ingeniero de software proviene de la complejidad arbitraria, forzada sin una razón por muchas instituciones y sistemas a los que se debe ajustar la interfaz. Esto puede diferir de interfaz a interfaz, o de un momento al otro, no necesariamente por necesidad, sino porque fueron designadas por diferentes personas.

En la mayoría de los casos, el software debe ajustarse porque es lo más reciente que ha llegado a la escena. En otros casos, debe hacerlo porque parece lo más correcto. Pero en casi todos los casos, la mayoría de la complejidad viene del tener que ajustarse a otros interfaces; esta complejidad no puede simplificarse solamente con un rediseño del software.

**Cambiabilidad:** El software está sujeto a una presión constante por el cambio. Esto también ocurre con los vehículos, edificios o computadoras. Pero los objetos manufacturados son raramente cambiados luego de su producción, los cambios vienen en modelos siguientes, o algunos cambios esenciales pueden ser incorporados en las tandas más recientes para los mismos diseños básicos. Las modificaciones de los productos manufacturados son mucho menos frecuentes que los cambios del software en producción.

Parte de esto se debe a que el software dentro de un sistema se adapta a su función, y la función es la parte que más siente las presiones por el cambio. En parte es porque el software puede ser cambiado de forma más fácil, es infinitamente maleable. Los edificios se pueden cambiar, pero los altos costos del cambio, entendidos por todos, sirven para desalentar estos cambios.

Todo software exitoso cambia. Cuando un producto de software resulta ser útil, la gente busca nuevas formas de llevarlo al límite, o más allá, del dominio original. Las presiones por extender las funciones vienen de parte de los usuarios a los que les gustan las funciones básicas e inventan nuevos usos para ellas.

Por otro lado, todo software exitoso sobrevive al ciclo de vida de la máquina en donde se despliega originalmente. Si no es la máquina entera, al menos debe sobrevivir a cambios en diferentes partes o periféricos de la misma.

En resumen, los productos de software están embebidos en una matriz cultural de aplicaciones, usuarios, leyes y máquinas, que están constantemente sujetas al cambio, y sus cambios inexorablemente llevan al cambio en los productos de software.

**Invisibilidad:** El software es invisible. Las abstracciones geométricas son herramientas poderosas, como lo son los planos de una planta de un edificio, que ayudan a evaluar espacios, flujos de tráfico, ayudan a la visualización de problemas u omisiones.

La realidad del software inherentemente no está embebida en el espacio, por lo que no tiene representaciones geométricas como lo sería un plano. Cuando intentamos diagramar la estructura de un software, nos encontramos con que no constituye uno, sino muchos grafos dirigidos superpuestos. Estos grafos pueden representar flujos de control, flujos de datos, patrones de dependencia, secuencias de tiempo, entre otros. Estos general mente ni siquiera son planos, mucho menos jerárquicos. De hecho, una forma de establecer cierto control, es intentar que todo encaje dentro de un mismo, o varios, grafos jerárquicos.

Al pesar de los progresos restringiendo y simplificando las estructuras de software, siguen siendo inherentemente no visualizables. Esto no solo impide que el proceso de diseño se pueda llevar a cabo por una única persona, sino que hace que sea compleja la comunicación entre diferentes personas.

**Innovaciones que resolvieron dificultades accidentales**

Si se examinan los tres avances en tecnologías de software que han sido más fructíferas en el pasado, podemos ver que cada una atacó a una diferente dificultad mayor en la construcción de software, pero han sido dificultades accidentales, no esenciales.

* **Lenguajes de alto nivel:** Sin dudas, el avance más poderoso para el incremento en la productividad, fiabilidad y simplicidad del software ha sido la inclusión progresiva de los lenguajes de programación de alto nivel. Los lenguajes de alto nivel han liberado a los programas de una gran parte de su complejidad accidental. Un programa abstracto está constituido por construcciones conceptuales, como operaciones, tipos de datos, secuencias y comunicaciones. El programa de máquina concreto se preocupa de los bits, registros, condiciones, discos, y cosas así. Los lenguajes de alto nivel mantienen las construcciones deseadas para los programas abstractos y dejan de lado las de más bajo nivel, eliminando un nivel entero de complejidad que nunca fue inherente al software.  
  La mayoría de los lenguajes de alto nivel pueden lograr que el nivel de sofisticación por la que se preocupa el programador es sobre estructuras de datos, tipos de datos y operaciones, haciendo que el desarrollo de los lenguajes se aproxime cada vez más a la sofisticación de los usuarios.
* **Tiempo compartido:** El Tiempo Compartido consiguió una gran mejora en la productividad de los programadores y en la calidad de su producto, aunque no tan grande como los lenguajes de alto nivel.  
  El tiempo compartido ataca una dificultad bastante diferente. Permite la inmediatez, y esto nos permite tener una visión de conjunto de la complejidad. El lento desarrollo del trabajo por lotes implicaba que uno se olvidaba de los detalles que tenía en mente en el momento que se dejaba de programar y se preparaba para compilar y ejecutar. El efecto más serio era el dejar de tener en mente todo lo que está ocurriendo en un sistema complejo.
* **Entornos de desarrollo unificados:** Unix e Interlisp, fueron los primeros entornos de programación integrados que se utilizaron ampliamente, y han mejorado la productividad en un grado importante.  
  Atacan las dificultades accidentales de usar múltiples programas, al proveer librerías integradas, archivos con formatos unificados, colas y filtros.  
  Este avance estimuló la creación de “bancos de herramientas”, puesto a que cada herramienta podía aplicarse a cualquiera de los programas que utilizaban los formatos estandar.

**Esperanzas de la plata**

Ahora, consideraremos los avances técnicos que suelen ser tratados como potenciales balas de plara.

* **Ada y otros lenguajes de alto nivel:** Uno de los avances técnicos más promocionados fue el lenguaje de programación Ada, un lenguaje de propósito general de alto nivel de la década de 1980. Ada no solo refleja los avances revolucionarios respecto a los lenguajes, sino que también presenta características que incentivan a los conceptos de diseño y modularización. Quizás, la filosofía de Ada es un mayor avance que el lenguaje en si, debido a su filosofía de modularización, tipos de datos abstractos, estructuras jerárquicas. La estructuración avanzada de sistemas de software fue una muy buena forma de aprovechar las capacidades de los nuevos microprocesadores de la época. Si bien Ada no se consideró como una bala de plata para la productividad en el desarrollo de software, el mayor beneficio de estos lenguajes llegó la primera vez que se adoptaron, hubo una transición entre las complejidades accidentales de la máquina (dadas por los lenguajes de bajo nivel y el gran control que había sobre los recursos de hardware), y el uso de las instrucciones más abstractas. Una vez que estos “accidentes” fueron eliminados, los restantes serían más pequeños, al igual que las recompensas por solucionarlos.
* **Programación orientada a objetos:** Muchos estudiantes de las tecnologías tenían más esperanzas en la POO que en cualquier otra técnica que se estuviera desarrollando en el momento. Se remarcaba que había que ser cauteloso al momento de distinguir dos conceptos: tipos de datos abstractos y tipos jerárquicos, también llamados clases. Ambos conceptos representan grandes avances en el desarrollo de software.  
  Cada uno elimina una dificultad accidental más del proceso, permitiendo al diseñador expresar la esencia de su diseño sin la necesidad de expresar grandes cantidades de material sintáctico que no agrega nueva información. Tanto los tipos abstractos como jerárquicos permiten eliminar dificultades accidentales de “orden superior” permitiendo expresar los diseños a más alto nivel.  
  Sin embargo, estos avances no hacen más que quitar la dificultad accidental de expresar el diseño, sin embargo, la complejidad del diseño en sí es esencial, por lo que estos avances no hacen que eso se modifique.
* **Inteligencia artificial:** Muchos esperaban que los avances en inteligencia artificial pudieran proveer un avance revolucionario que produzca un salto en la productividad y calidad del software. Es difícil pensar en cómo las técnicas de inteligencia artificial, como el reconocimiento de imágenes, por ejemplo, puedan generar una diferencia apreciable en la programación. Lo mismo se puede decir de técnicas como las de reconocimiento del habla, lo difícil respecto al desarrollo de software es “decidir que decir, no decirlo”, siguiendo la analogía de las técnicas de reconocimiento del habla. Ninguna facilidad en forma de expresar el diseño puede dar más que una ganancia marginal.
* **Sistemas expertos:** La parte más avanzada de la inteligencia artificial, y la que tenía más ampliamente aplicada, es la tecnología para construir sistemas expertos.  
  Un sistema experto es un programa que contiene un motor de inferencia generalizado, y una base de reglas, diseñado para tomar datos de entrada y suposiciones, y explorar las consecuencias lógicas de las inferencias derivadas de la base de reglas, sacando conclusiones y recomendaciones. Generalmente los motores de inferencia pueden lidiar con datos y reglas difusos y probabilísticos, junto con la aplicación de pura lógica determinista.  
  Los sistemas expertos ofrecen ventajas sobre los algoritmos programados para llegar a las mismas soluciones, entre ellas, los motores de inferencia son desarrollados independientemente de la aplicación, y luego son aplicados a diferentes usos. Las partes que son particulares para cada aplicación son codificadas en la base de reglas en una forma unificada, por lo que existen herramientas para desarrollar, cambiar, probar y documentar las bases de reglas.  
  Un avance muy importante ofrecido por estas tecnologías es la separación de la complejidad de la aplicación de la programación en si.  
  La mayor contribución de los sistemas expertos puede ser que pone la experiencia y la sabiduría acumulada de los mejores programadores al servicio de los más inexpertos. La brecha que existe entre las mejores prácticas de ingeniería de software y las prácticas promedio es muy amplia, por lo que una herramienta que achique esta brecha pudo haber sido importante.
* **Programación “automática”:** Durante años, la gente se ha estado anticipando y escribiendo sobre la generación de un programa para resolver un problema a partir de la declaración de las especificaciones del problema, algunos esperan que esta tecnología produzca el siguiente gran avance.
* **Programación gráfica:** Un tema muy popular para las tesis de doctorados en ingeniería de software era la programación gráfica o visual, que es la aplicación de gráficos de computadora para el diseño de software. A veces, esta técnica es justificada al considerar los diagramas de flujo como el medio ideal para el diseño de programas, proveyendo poderosas comodidades para construirlos.  
  A pesar de las expectativas, nada ha surgido para que esto sea viable, principalmente debido a que los diagramas de flujo son una abstracción muy pobre de la estructura del software.  
  Además, como se mencionó anteriormente, el software es muy difícil de visualizar, sin importar el tipo de diagrama que se utilice, siempre se estará visualizando solo única dimensión del gran problema, y si superponemos múltiples diagramas, generados a partir de múltiples vistas relevantes es muy difícil extraer una vista global.
* **Verificación de programas:** Gran parte del esfuerzo en el desarrollo de software va destinado a las pruebas y la corrección de errores. Entonces, quizás puede existir una bala de plata que permita eliminar los errores en la fase de diseño? Se pueden mejorar radicalmente la productividad y la fiabilidad del producto diseñando el producto correctamente antes de realizar el gran esfuerzo de implementarlo y probarlo?  
  La verificación de los programas es una herramienta muy potente, y lo va a seguir siendo. La tecnología no promete ahorrar trabajo, las verificaciones son tas costosas que solo unos pocos programas son verificados.  
  Verificar los programas no significa que van a estar libres de fallos, no hay ninguna magia aquí. Si bien la verificación de los programas puede reducir la carga de las pruebas, no las puede eliminar.  
  Hasta una verificación perfecta puede establecer solamente que un programa cumple con su especificación. La tarea más dura en el software es conseguir unas especificaciones completas y consistentes, y la parte más esencial de construir un programa es, de hecho, la depuración de las especificaciones.
* **Entornos y herramientas:** Las mejoras que se pueden obtener a partir de la inversión en obtener mejores entornos de programación son pequeñas. Si bien mejores entornos de desarrollo aumentan la productividad y la fiabilidad, los problemas más grandes aparentan estar resueltos, por lo que el retorno de las inversiones en mejores entornos serán marginales.
* **Estaciones de trabajo:** Las estaciones de trabajo actuales tienen la capacidad suficiente para la edición de los programas, y aunque las tareas de compilación podrían incrementar su velocidad con mayor potencia computacional, lo que dejaría que la principal actividad en el día de un programador sea el tiempo para pensar. Una mayor potencia en las estaciones de trabajo siempre es bienvenida, pero no se puede esperar una solución mágica a partir de ello.

**Ataques prometedores a la Esencia Conceptual**

Todos los ataques tecnológicos hacia los “accidentes” del proceso de software están limitados por la ecuación de productividad:



Si los componentes conceptuales de la tarea llevan la mayor parte del tiempo, entonces ninguna actividad sobre las partes necesarias del trabajo que consistan simplemente en la expresión de los conceptos dará grandes ganancias de productividad.

Deberíamos centrarnos en aquellos ataques sobre la esencia del problema del software, la formulación de estructuras conceptuales complejas. Afortunadamente, algunos de estos ataques son prometedores.

* **Comprar vs Desarrollar:** La solución más radical posible a la construcción del software es no construirlo. Cada día esto se torna más fácil a medida que más y más vendedores ofrecen más y mejores productos de software para una gran variedad de aplicaciones. La compra de cualquier producto de software siempre es más barata que desarrollarlo, y se obtiene inmediatamente, al menos para productos que ya existen. Estos productos suelen estar mejor documentados y hasta mejor mantenidos que los programas desarrollados a medida.  
  El costo del software ha sido siempre el costo de su desarrollo , por lo que dividir esos costos incluso en unos pocos usuarios, disminuye radicalmente los costos por usuario.   
  Por supuesto, el tema clave es la aplicabilidad, decidir si se puede utilizar un paquete de software comercial para las tareas que necesito. Esto depende de lo generalizados y personalizables que sean los paquetes.  
  Parte de la decisión sobre elegir un paquete de software comercial o un paquete de software a medida cae en el uso que se le dará y el hardware en el que se ejecutará, sobre todo en la relación de costo hardware/software. Quizás una empresa que debe gastar $2M de dólares en equipamiento esté dispuesta a gastar $250000 en el desarrollo de un paquete de software a medida, pero aquellos que deban comprar equipos por $50000, no podrán concebir la idea de pagar un software hecho a medida, por lo que eligen adaptarse a paquetes de software comerciales.
* **Refinamiento de requerimientos y prototipado rápido:** La parte individual más difícil de un sistema de software es decidir con precisión que es lo que hay que construir. Ninguna otra parte del trabajo conceptual es tan difícil como establecer detalladamente los requerimientos, incluyendo todas las interfaces para personas, máquinas y otros sistemas de software. Ninguna otra parte del trabajo invalida el sistema resultante si se hace mal, ninguna otra parte es más difícil de corregir en estados avanzados.  
  Por lo tanto, la función más importante que los desarrolladores de software cumplen para sus clientes es la extracción y refinamiento iterativos de los requerimientos del producto. Los clientes no saben lo que quieren, generalmente no conocen que preguntas se deben responder, y casi nunca han pensado en detalle sobre el problema que necesitan especificar. Los sistemas de software complejo son difíciles de imaginar, por lo que para planificar cualquier actividad de diseño de software, es necesario permitir un trabajo iterativo extensivo entre el cliente y el diseñador como parte de la definición del sistema.  
  Cabe resaltar que es imposible para un cliente, trabajando con un ingeniero de software, especificar completa, precisa y correctamente los requisitos exactos de un sistema de software sin probar algunas versiones de prueba del sistema.  
  Por lo tanto, una de las mayores promesas de los esfuerzos tecnológicos actuales, que ataca a la esencia, no a los accidentes, del problema de software es el desarrollo de herramientas para el prototipado rápido de sistemas, ya que el prototipado es parte de la especificación iterativa de requisitos.  
  Un prototipo es un software que simula las interfaces importantes del sistema y lleva a cabo las funciones principales del sistema buscado, pero no necesariamente se adecúa a las necesidades de rendimiento o al hardware necesario, generalmente llevan a cabo las tareas principales de la aplicación pero sin preocuparse por el manejo de excepciones, la validación de las entradas, cierres seguros, etc. El propósito de un prototipo es convertir la estructura conceptual especificada en algo real, para que el cliente pueda probar su usabilidad.
* **Desarrollo incremental, hacer crecer, no construir software:** Ocurrió un cambio muy grande a medida que se dejó de hablar de “escribir software” y se empezó a hablar sobre “construir software”, el cambio de metáfora fue poderoso y preciso, hoy en día entendemos las grandes similitudes que hay entre la construcción de software y cualquier otro proceso de construcción, pudiendo usarse otros elementos de la metáfora como la especificación, o el ensamblado de los componentes.  
  Cambiando de analogía, pensemos en el estudio de los seres vivientes. El cerebro por si mismo es más intrincado que lo que se puede ver en un mapeo, más poderoso que cualquier imitación, capaz de protegerse a si mismo y auto-renovable. El secreto de esto está en la evolución o incremento del mismo, no en su construcción.  
  De la misma forma debería ser con los sistemas de software, deberían crecer con un desarrollo incremental. Esto significa que, el sistema al principio debería realizarse para que se ejecute, aunque no haga nada más que simplemente llamar al conjunto correcto de subprogramas, a partir de allí, deberá ser desarrollado implementando las funciones correspondientes a partir de los subprogramas. Este enfoque ha incrementado cambiado dramáticamente la eficacia del desarrollo de software. Permite rastreas fácilmente los cambios (backtracking), permite la obtención de prototipos tempranos, cada función añadidad y cada nueva provisión para datos o circunstancias más complejos crece organicamente a partir de lo ya existente.
* **Excelentes diseñadores:** La cuestión principal sobre cómo mejorar el arte del software se centra en las personas. Se pueden tener buenos diseños siguiendo buenas prácticas. Las buenas prácticas de diseño se pueden aprender.  
  Mientras que la diferencia entre los malos diseños conceptuales y los buenos recae en la robustez del método de diseño, la diferencia entre los buenos diseños y los excelentes seguro que no. Los diseños excelentes provienen de excelentes diseñadores, la construcción de software es un proceso creativo. Una metodología robusta puede empoderar y liberar a una mente creativa, pero no puede inspirar a quien no lo es.  
  Las diferencias no son menores, los estudios muestran que los mejores diseñadores producen estructuras que son más rápidas, pequeñas, simples, limpias y producidas con un menor esfuerzo. La diferencia entre un enfoque excelente y uno promedio puede ser un salto muy grande.  
  Si bien muchos sistemas de software refinados y útiles han sido diseñados por comités y construidos por proyectos multipartes, los sistemas de software que han exaltado a los fanáticos son aquellos que provienen de uno o unos pocos diseñadores, excelentes diseñadores.  
  A partir de esto, si bien la transición de tecnologías es importante, el esfuerzo más importante que se debe hacer es buscar formas de cultivar excelentes diseñadores.  
  Tanto los excelentes diseñadores, como excelentes administradores son muy raros, por eso la mayoría de las organizaciones se esfuerzan por encontrar y cultivar a los mejores administradores, pero pocas dedican el mismo esfuerzo en encontrar y desarrollar excelentes diseñadores, de los cuales va a depender la excelencia técnica de los productos.  
  Las empresas de software deben determinar que los diseñadores excelentes son tan importantes para su éxito como lo son los administradores, y que deben ser recompensados de formas similares, no solo en salario, sino también en otros requisitos como equipamiento técnico personal, personal de soporte, fondos para viajes, etc.  
  Algunos pasos obvios para el desarrollo de excelentes diseñadores:
  + Identificar sistemáticamente los mejores diseñadores tan pronto como sea posible. Los mejores no siempre son los más experimentados.
  + Asignarles un mentor de carrera que sea responsable de su desarrollo.
  + Idear y mantener un plan de desarrollo de carrera para cada una de las “promesas”, incluyendo aprendizaje junto a otros diseñadores excelentes cuidadosamente seleccionados, episodios de educación formal avanzada y cursos cortos, todo mezclado con tareas de diseño individuales y de liderazgo técnico.
  + Brindar oportunidades para que los diseñadores en formación interactúen entre ellos.

User Stories Primer

Para los propósitos del estudio, se definirá una user story como:

Una user story es una breve declaración de una intención que describe algo que el sistema necesita hacer para el usuario.

En XP, generalmente es el cliente el que escribe las user stories, por lo que es necesario incluirlo directamente en el proceso de desarrollo. En Scrum, es el product owner quien se encarga de escribirlas, con entradas directas desde el cliente, el equipo y los stakeholders. Sin embargo, cualquier miembro del equipo con suficiente conocimiento sobre el dominio puede escribir las user stories, pero es el product owner quien debe aceptarlas y priorizarlas para incluirlas en el prodct backlog.

Las user stories son una herramienta para describir el comportamiento del sistema de una forma que sea entendible tanto para los desarrolladores como para los usuarios. Las historias de usuario se enfocan en el valor definido por el usuario, en vez de una estructura funcional, la cual es la forma en la que tradicionalmente se realiza el trabajo. Las historias de usuario proveen una enfoque liviano y efectivo para administrar los requerimientos de un sistema.

Una historia de usuario captura una declaración corta de una funcionalidad en una “index card”.

Los detalles del comportamiento del sistema no deben aparecer en esta declaración breve, se deben dejar para ser desarrolladas en futuras conversaciones y criterios de aceptación entre el equipo de desarrollo y el product owner.

**Las historias de usuario ayudan a achicar la brecha de cominicación entre desarrolladores y clientes**

En los desarrollos ágiles, es el trabajo de los desarrolladores hablar el lenguaje de los usuarios, no el del usuario hablar el lenguaje tecnológico. La comunicación efectiva es la clave, y se necesita un lenguaje común. Las historias de usuario proveen este lenguaje común que facilita el entendimiento entre el usuario y el equipo técnico.

Con las historias usuario no necesitamos entender el lenguaje del otro a un gran nivel, solo necesitamos entendernos lo suficiente como para saber cuando hemos realizado una buena negociación.

**Las historias de usuario no son requerimientos**

Mientras las historias de usuario hacen la mayoría del trabajo que se hacía con las especificaciones de requerimientos de software, casos de uso y herramientas similares. Son materialmente diferentes en puntos sutiles pero críticos.

* No hay especificaciones detalladas de requerimientos, sino que son expresiones negociables de una intención (es algo que se debe abordar).
* Son cortas y fáciles de leer, entendibles para desarrolladores, stakeholders y usuarios.
* Representan pequeños incrementos en funcionalidad con valor, que se puede desarrollar en un periodo de días a semanas.
* Son relativamente fáciles de estimar, por lo que el esfuerzo de implementar la funcionalidad puede ser determinado rápidamente.
* No deben ser mantenidas a largo plazo, a diferencia delos documentos, pero deben ser organizadas en listas que pueden ser más fáciles de organizar a medida que se obtiene más información.
* No se detallan al comienzo del proyecto, pero son elaboradas sobre la base de “just-in-time”, evitando especificaciones muy tempranas, retrasos en el desarrollo, el inventario de requerimientos, y declaraciones demasiado restringidas sobre la solución.
* Necesitan poco o nada de mantenimiento, y pueden ser descartadas de forma seguro tras su implementación.
* Las historias de usuario y el código creado a partir de ellas sirven como entradas para la documentación, la cual también es construida incrementalmente.

**Card, Conversation and Confirmation**

Ron Jeffries utiliza Tarjeta, Conversación, y Confirmación para describir los tres elementos de una historia de usuario:

**Tarjeta:** Se representa con 2 o 3 oraciones usadas para definir la intención de la historia. La tarjeta sirve como un token de recuerdo, que resume la intención y representa un requerimiento más detallado, cuyos detalles deben ser determinados.

**Conversación:** Representa una discusión entre el equipo, cliente, product owner y otros stakeholders, la cual es necesaria para determinar de forma más detallada el comportamiento requerido para implementar la intención. En otras palabras, la tarjeta representa una “promesa para una conversación” sobre la intención.

**Confirmación:** Representa la prueba de aceptación, la cuál es la forma en que el cliente o el product owner confirmarán que la historia ha sido implementada satisfactoriamente para ellos. En otras palabras, la confirmación representa las condiciones de satisfacción que serán aplicadas para determinar si la historia cumple o no la intención.

Con esto, tenemos una muestra de como la calidad en entornos ágiles es obtenida durante el desarrollo del código. Esto se logra simplemente asegurándonos de que cada nueva historia de usuario es discutida y refinada en cualquier detalle que sea necesario, y, testeando la implementación según la satisfacción de los stakeholders claves.

**Voz de historia de usuario:**

En los últimos años se ha aplicado una nueva forma estándar que fortalece significativamente la estructura de la historia de usuario:

**Como** <rol> **quiero/necesito** <actividad> **para** <valor de negocio>

Donde:

* **Rol:** Representa quien está realizando la acción, o quizás uno de los que está recibiendo el valor de la actividad. Puede ser otro sistema si es lo que está instanciando la actividad.
* **Actividad:** Representa la acción que será realizada por el sistema.
* **Valor de negocio:**

Llamamos a esta forma de expresar la historia de usuario como “voz de usuario”, y ha resultado ser una construcción muy útil porque separa el espacio del problema (el valor de negocio entregado/requerido) y el espacio de la solución (actividad, lo que hace el sistema). También provee una perspectiva “user-first” para el equipo, que lo mantiene enfocado en el valor de negocio y resolver problemas reales para personas reales.

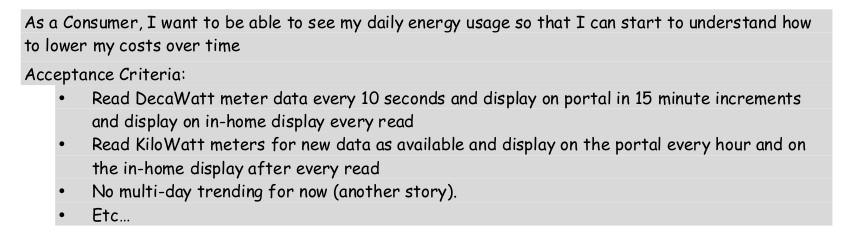
Esta forma de expresar las historias de usuario mejoran la comprensión de por qué, y cómo los desarrolladores deben implementar un sistema que realmente satisfaga las necesidades de los usuarios.

**Detalles de historia de usuario**

Los detalles de una historia de usuario son convenidos principalmente a través de conversaciones entre el product owner y el equipo, manteniendo al equipo involucrado desde el principio. De todas formas, si se necesitan más detalles sobre la historia, pueden ser provistos en la forma de agregados (bocetos, hojas de cálculo, algoritmos o lo que sea), las cuales se adjuntan a la historia de usuario. En ese caso, la historia de usuario sirve como un “token” el cual también acarrea el comportamiento más específico para el equipo.  Los detalles adicionales deben ser recolectados a lo largo del tiempo (just-in-time) a través de discusiones y colaboraciones con el equipo y otros stakeholders, antes y durante el desarrollo.

**Criterios de Aceptación de historias de usuario**

En adición de la declaración de la historia de usuario, se pueden mantener notas adicionales, asunciones y criterios de aceptación junto a la historia de usuario. Muchas conversaciones sobre la historia de usuario se llevarán a cabo entre el equipo y clientes antes y durante el momento en que la historia se pase a código. Los flujos alternos en la actividad, los límites de aceptación y otras aclaraciones deben ser capturadas junto con la historia. Muchas de estas pueden ser convertidas en pruebas de aceptación, u otros casos de pruebas funcionales, para la historia de usuario.



Los criterios de aceptación no son pruebas funcionales ni unitarias, en lugar de eso son condiciones de satisfacción para el sistema. Las pruebas funcionales y unitarias abarcan pruebas más profundas a todos los flujos funcionales, flujos de excepciones, condiciones de límites y funcionalidades similares asociadas a la historia.

**INVEST** (invertir) en buenas historias de usuarios

Los equipos ágiles pasan una buena cantidad de tiempo (quizás la mitad o hasta más) descubriendo, elaborando y entendiendo las historias de usuario, y escribiendo las pruebas de aceptación para ellas. Esto debe ser así porque representa el hecho de qué:

*Escribir el código para objetivos entendidos no es necesariamente la parte más difícil del desarrollo de software, sino que más bien lo es el entender cuales son los objetivos reales para el código.*

Por lo que, invertir en buenas historias de usuario, aunque sea al último momento, es un esfuerzo que vale la pena para el equipo. Bill Wake estableció el acrónimo INVEST, para describir los atributos de una buena historia de usuario.

* Independent (independiente).
* Negotiable (negociable).
* Valuavle.
* Estimable.
* Small (pequeña).
* Testable (testeable).

El modelo INVEST es ubicuo y muchos equipos ágiles evalúan sus historias para que representen estos atributos.

**Independiente:** Significa que una historia de usuario puede ser desarrollada, testeada, y potencialmente incluso ser entregada por sí misma. Por lo que también debe ser tener un valor independiente.

Muchas historias de usuario van a tener alguna dependencia secuencial natural como producto de la construcción de las funcionalidades, de todas formas, cada historia debe entregar su valor independientemente.

De cualquier forma, muchas dependencias que no aportan valor por si mismas, ya sean técnicas o funcionales, también tienden a buscar su camino dentro de los backlogs y éstas necesitan ser encontradas y eliminadas.

**Negociable:** A diferencia de los requerimientos tradicionales, una historia de usuario no es un contrato para una funcionalidad específica, más bien es un recordatorio de requerimientos que necesitan ser discutidos., desarrollados, probados y aceptados. Este proceso de negociación entre el negocio y el equipo reconoce la legitimidad y primacía de las entradas desde el negocio, pero permite que sean descubiertas a trabes de la colaboración y retroalimentación.

Anteriormente, con las organizaciones aisladas, los requerimientos escritos eran requeridos generalmente para facilitar la comunicación limitada entre los departamentos, y para servir como registro de acuerdos pasados. Ágil, sin embargo, está fundado en el concepto de que un enfoque basado en equipo es más efectivo para resolver problemas en un entorno colaborativo dinámico. Una historia de usuario se realiza en tiempo real y es estructurada para aprovechar este enfoque de comunicación y colaboración directa y efectiva.

**Valuable:** El objetivo de un equipo ágil es simple: entregar el mayor valor teniendo en cuento las limitaciones de tiempo y recursos existentes. Por lo que, el Valor es el atributo más importante en el modelo INVEST, y cada historia de usuario debe otorgar algún valor al usuario, cliente o stakeholder del producto. Los backlogs son priorizados por valor, y el éxito o fracaso del negocio está basado en el valor que los equipos pueden entregar.

Un desafío típico al que se enfrentan los equipos es aprender a cómo escribir historias de usuario pequeñas e incrementales que puedan entregar valor efectivamente.

Las historias de usuario buscan una perspectiva vertical, en vez de capas horizontales. Se puede pensar una historia de usuario como una torta multicapa, por ejemplo, una capa de red, una de persistencia, lógica y presentación. Cuando se divide una historia horizontalmente, estamos sirviendo solo una parte de esa torta, cuando en realidad queremos darle al cliente la esencia de la torta completa, y la mejor forma de hacerlo es rebanar verticalmente a través de las capas. Los desarrolladores a menudo tienen la inclinación de trabajar en una única capa a la vez, pero una capa completa de bases de datos (por ejemplo) tiene poco valor para el cliente si no hay una capa de presentación.

Crear historias con valor requiere que reorientemos nuestras estructuras funcionales de un enfoque horizontal a uno vertical. Creamos historias de usuario que atraviesan la arquitectura para que podamos presentar valor al usuario y buscar su devolución tan pronto y seguido como sea posible.

Mientras que normalmente el valor está enfocado en la interacción del usuario con el sistema, a veces el valor es más apropiado si se enfoca en una vista de el cliente o stakeholders claves.

**Estimable:** Una buena historia de usuario es estimable.. Mientras que en el backlog pueden haber historias de cualquier tamaño, en orden de que puedan ser desarrolladas y probadas en una iteración, el equipo debería ser capaz de proveer una estimación aproximada de su complejidad y la cantidad de trabajo requerida para completarla. El mínimo esfuerzo de estimación es para determinar si se puede completar en una única iteración. La precisión adicional en la estimación le dará al equipo una mayor predictibbilidad.

Si el equipo no es capaz de estimar una historia de usuario, generalmente indica que la historia es muy grande o que tiene mucha incertidumbre. Si es muy grande para estimar, se debería dividir en historias más pequeñas. Si la historia tiene mucha incertidumbre para estimar, entonces se puede usar una spike técnica o funcional para reducir la incertidumbre, para que una o más historias de usuario estimables surjan como resultado.

Uno de los principales beneficios de estimar historias de usuarios no es simplemente obtener un tamaño preciso, sino aclarar cualquier asunción oculta, criterios de aceptación perdidos, y para que el equipo tenga un entendimiento compartido de la historia. Por lo que la conversación alrededor del proceso de estimación es tan importante (o más) que la estimación en sí. La habilidad de estimar una historia de usuario está altamente influenciada por la capacidad de la historia.

**Small:** Las historias de usuario deben ser lo suficientemente pequeñas para ser completadas en una iteración, de otra forma no podrán proveer valor alguno o ser consideradas “hechas” para ese momento. De todas formas, incluso las historias de usuario más pequeñas proveen más agilidad y productividad. Hay dos principales razones para esto: rendimiento incrementado (cantidad de trabajo realizado por unidad de tiempo) y complejidad disminuida.

Al tener un rendimiento constante, es necesario reducir la carga de trabajo para reducir el tiempo de ciclo (tiempo en que se realizan las tareas). Esto quiere decir, que utilizando historias pequeñas estas se terminarán más rápido.

Las historias más pequeñas no solo se realizan más rápido por su tamaño en si, sino que se llevan a cabo más rápido principalmente por su complejidad reducida, y la complejidad tiene una relación no lineal (exponencial) con su tamaño (por eso para estimar se utilizan series como la de Fibonacci). Esto se vé más fácilmente en las pruebas, donde cantidad de pruebas necesarias para validar la funcionalidad incrementa a un ritmo exponencial con la complejidad de la función en sí. Esto se relaciona con tres reglas de Robert Martin sobre escribir código simple:

* Regla 1: Hacer una cosa.
* Regla 2: Mantenerla pequeña.
* Regla 3: Hacerla incluso más pequeña.

Existe una relación entre tamaño e independencia. Las historias de usuario pequeñas incrementan el número de dependencias entre ellas, sin embargo, las historias pequeñas, incluso con el incremento de las dependencias, entregan un mayor valor y proveen retroalimentaciones más rápido por parte del usuario que las historias grandes. Por eso, el siempre hay que inclinarse por hacer historias pequeñas, y luego hacerlas incluso más pequeñas.

**Testable:** En ágil propiamente realizado, todo el código es código testeado, por lo que se deduce que todas las historias deben ser testeables. Si una historia de usuario no aparenta ser testeable, entonces la historia probablemente está mal formada, demasiado compleja, o quizás depende de otras historias en el backlog.

Para asegurar que una historia no entre en una iteración en la que no pueda salir (siendo completamente testeada) muchos equipos ágiles toman un enfoque “write-the-test-first”. Esto comenzó en la comunidad de XP con Test Driven Development (TDD), la práctica de escribir pruebas unitarias automatizadas antes de escribir el código.

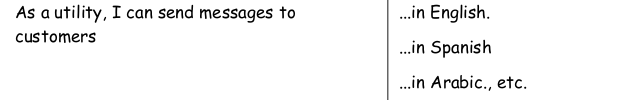
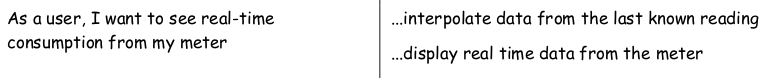
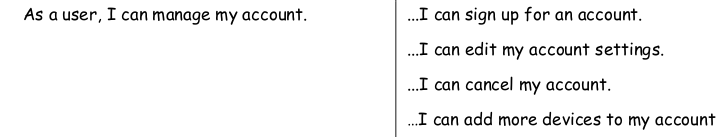
Según esta filosofía, si un equipo realmente sabe como testear una historia, entonces prubablemente sepan como implementarla también.

Para asegurar la capacidad de ser testeadas, las historias de usuario comparten ciertos criterios con los requerimientos. Las palabras como “rápido”, “lindo”, “limpio”, etc. son fáciles de escribir, pero difíciles de probar porque significan diferentes cosas para diferentes personas, por lo que se deben evitar. Mientras estas palabras proveen negociabilidad, enmarcarlas con límites claros puede ayudar al equipo y al negocio a compartir las expectativas del resultado y evitar grandes sorpresas.

**Dividiendo historias de usuario**

Una historia de usuario generalmente comienza como una “feature” o una épica (un amplio, vago concepto de algo que queremos hacer para el usuario). A menudo encontramos estas historias de gran valor durante el proceso de descubrimiento y las capturamos en el backlog. Sin embargo, estas son historias compuestas, y generalmente son muy grandes para ser implementadas en una iteración. Para preparar el trabajo para las iteraciones, el equipo debe dividirlas en historias más pequeñas,

No hay una rutina para separar las historias de usuario en partes aptas para una iteración más allá de la regla general de hacer que cada historia cruce verticalmente a la arquitectura, y agregue valor al usuario. Sin embargo, se recomienda aplicar una apropiada selección de entre 10 patrones comunes para dividir una historia de usuario.

1. Pasos de flujo de trabajo: Identificar los pasos específicos que debe seguir el usuario para completar un flujo de trabajo específico e implementarlo incrementalmente.
2. Variaciones de reglas de negocio: A primera vista, algunas historias parecen bastante simples. Sin embargo, a veces las reglas de negocio son mas complejas o extensas de lo que se vé a primera vista. En este caso, puede ser útil partir la historia en múltiples historias que manejen la complejidad de la regla de negocio.
3. Mayor esfuerzo: A veces las historias de usuario se pueden dividir en partes en donde la mayor parte del esfuerzo se dedicará a implementar la primera, mientras que agregar más funcionalidades a partir de ella debería ser relativamente trivial.
4. Simple/Complejo: Mientras el equipo está discutiendo una historia, y ésta parece que se hace cada vez más grande, se debe parar y preguntar “¿Cuál es la forma más simple en que puede funcionar?”. Capturar la versión más simple como una historia separada, y a partir de ahí separar todas las variaciones y complejidades en historias individuales.
5. Variaciones en los datos: Las variaciones en los datos y en sus fuentes son otra fuente de complejidad. Se debe considerar agregar historias “just-in-time” después de realizar la versión más simple. Por ejemplo, un caso de cambios de idioma:  
     
     
     
     
   
6. Métodos de ingreso de datos: A veces, la complejidad está en las interfaces de usuario en lugar de la funcionalidad en sí. En este caso, se debe separar la historia para construirla con la interfaz de usuario más simple posible, y construir una UI más rica más adelante.
7. Aplazar características del sistema: A veces, la implementación inicial no es muy difícil, y la mayor parte del esfuerzo está en hacerla más rápida, confiable, precisa o escalable. Sin embargo, el equipo puede aprender mucho a partir de la implementación base, que debería darle valor al usuario, pero no debe ser capaz de hacerlo todo. En este caso, dividir la historia de usuario en “enfermedades” sucesivas.  
     
     
     
   
8. Operaciones (Ejemplo, CRUD): Palabras como administrar, o controlar son un un indicio de que la historia cubre múltiples operaciones, lo que puede ofrecer una forma natural de difidir a la historia:  
     
     
     
     
     
     
     
   
9. Escenarios de casos de uso: Si se han utilizado casos de uso para representar interacciones complejas user-to-system o system-to-system, entonces la historia a menudo se puede difidir de acuerdo a los escenarios individuales del caso de uso.
10. Deshacerse de una Spike: A veces una historia puede ser muy grande, muy compleja, o quizás su implementación se ha entendido de manera muy pobre. En ese caso, se debe construir una spike técnica o funcional para solucionarlo, luego dividir las historias en base a los resultados.

Cuando se dividen historias, el equipo debe utilizar una combinación apropiada de las técnicas anteriores para considerar medios de descomposición o múltiples patrones en combinación.

**Spikes**

Las spikes son otro invento de XP. Son un tipo especial de historias utilizadas para eliminar los riesgos e incertidumbres en una historia de usuario u otro aspecto del proyecto. Las spikes pueden utilizarse por diferentes razones:

* El equipo puede que no tenga conocimientos sobre un nuevo dominio, y las spikes pueden utilizarse para una investigación básica para familiarizar al equipo con una nueva tecnología o dominio.
* La historia puede ser muy grande para ser estimada apropiadamente, y el equipo necesita utilizar una spike para analizar el comportamiento implicado, y poder dividir la historia en partes estimables.
* La historia pueden tener riesgos técnicos significativos, y el equipo puede necesitar hacer una investigación o prototipos para obtener confianza en un enfoque tecnológico que les permita entregar la historia de usuario en un futuro.
* La historia puede contener riesgos funcionales significativos, aunque la intención de la historia pueda estar bien entendida, puede que no sea clara la forma en la que el sistema necesita interactuar con el usuario para lograr el beneficio.

**Spikes técnicas y funcionales**

Las spikes técnicas se utilizan para investigar varios enfoques técnicos para el dominio de la solución. Por ejemplo, una spike técnica se puede utilizar para determinar una decisión sobre construir vs comprar, o para evaluar la implementación de tecnologías específicas que se puedan aplicar a la solución.

Las spikes funcionales se utilizan siempre que existe una incertidumbre significativa respecto a como un usuario debe interactuar con el sistema. Las spikes funcionales a menudo se evalúan mejor a través de cierto grado de prototipado, ya sea con bocetos de la interfaz de usuario o cualquier técnica que sirva mejor para obtener feedback por parte del cliente o los stakeholders. Algunas historias de usuario pueden necesitar ambos tipos de spikes.

**Guiás para utilizar spikes**

Estimables, demostrables y aceptables.

Como otras historias, las spikes se deben incluir en el backlog y deben ser estimables y de un tamaño que se ajuste a una iteración. Los resultados de una spike son diferentes a los de una historia de usuario porque generalmente producen información en lugar de código funcionando. Una spike puede terminar en una decisión, un prototipo, una “storyboard”, una prueba de concepto, o en alguna otra solución parcial que ayude a obtener el resultado final. En cualquier caso, la spike debe desarrollar solo la información suficiente para resolver la incertidumbre para identificar y estimar las historias que se encuentran detrás de ella.

La salida de una spike es demostrable para el equipo. Brinda visibilidad sobre la investigación y los esfuerzos arquitectónicos, y ayuda a que se comparta la responsabilidad sobre las decisiones claves que se están tomando.

Y, como cualquier otra historia, las spikes son aceptadas por el product owner cuando los criterios de aceptación de la spike se han completado.

**La excepción, no la regla**

Toda historia de usuario tiene su incertidumbre y sus riesgos, es la naturaleza del desarrollo ágil. El equipo descubre la solución correcta a través de discusiones, colaboración, experimentación y negociación. Por lo que, de cierta forma, toda historia de usuario tiene actividades que corresponden a una spike para eliminar los riesgos técnicos y funcionales. El objetivo de un equipo ágil es aprender como aceptar y atacar efectivamente esta incertidumbre en cada iteración. Una spike, por otro lado, debe ser reservada para incertidumbres más grandes y críticas.

Cuando se considere una spike, primero se deben considerar formas de dividir a la historia a través de las estrategias discutidas anteriormente y usar una spike como última opción.

**Considerar implementar en un sprint separado a las historias resultantes de la spike**

Dado que una spike representa incertidumbre en una o más historias potenciales, planear la spike y la historia resultante en una misma iteración es riesgoso, y generalmente se debería evitar. Sin embargo, si la spike es pequeña y directa, y es probable que se encuentre una solución rápida, entonces no está mal completar las historias en la misma iteración, solo se debe ser cuidadoso.

**Modelado de historias con “Index Cards”**

Escribir las historias de usuario y modelar el valor entregado visualmente utilizando fichas físicas provee medios visuales cinéticos para comprometer al equipo completo en el desarrollo del backlog. Hay múltiples ventajas sobre este enfoque interactivo:

El tamaño físico de las fichas fuerzan a limitar la longitud del texto, haciendo que quien escribe la historia deba articular las ideas solo en una o dos frases. Esto ayuda a mantener las historias de usuario pequeñas y centradas. También la naturaleza física y tangible de las tarjetas le da al equipo la habilidad de ordenarlas visual y espacialmente de distintas formas para ayudar a definir el backlog.

* Las fichas se deben organizar por características (o épicas, o temas) y pueden ser escritas en tarjetas del mismo color las historias con las mismas características para ayudar a la diferenciación visual.
* Las fichas pueden ser organizadas por tamaño físico para ayudar a los desarrolladores a visualizar las relaciones de tamaño entre diferentes historias.
* Las fichas se pueden organizar por tiempo o por iteración para evaluar las dependencias, entender la secuencia lógica, observar el impacto en la velocidad del equipo y para comunicar y alinear de una mejor forma las prioridades contradictorias de los stakeholders.

Cualquier miembro del equipo puede escribir las fichas, y el acto físico de mover estos pequeños, tangibles “objetos de valor” sobre la mesa crea una cinética interactiva en donde los participantes “ven y tocan” el valor que están por crear para los stakeholders.

**Técnicas de Estimación**

Mientras más esfuerzo dedicamos a algo, mejor serán los resultados, pero a veces solo necesitamos una fracción de ese esfuerzo para obtener resultados adecuados.

Necesitamos mantenernos conscientes de la disminución en el retorno del tiempo que invertimos estimando a medida que dedicamos una mayor cantidad de tiempo a esa tarea. A menudo pasamos un poco de tiempo pensando en una estimación y obteniendo un número que es casi tan bueno como si hubiéramos invertido mucho tiempo en ello.

Hay que variar el esfuerzo puesto en la estimación de acuerdo al propósito de la estimación. Si se está tratando de estimar algo relativamente simple, no vale la pena dedicar demasiado esfuerzo a estimarlo. En cambio, si se debe estimar una tare más compleja, es necesario dedicar una mayor cantidad de tiempo a la estimación, pero tampoco al punto de pasar la mitad del tiempo de la duración de la tarea estimando la misma.