USER STORIES

Libro: "*A user story primer" (Introducción a las historias de usuario)*

Introducción

Una user story es un liviano y más ágil sustituto de lo que ha sido la forma tradicional de especificar requerimientos de software. Es el contenedor que primeramente lleva el flujo de valor al usuario, y el desarrollo ágil se trata todo acerca de la entrega de valor rápida. Junto con los Temas, las Épicas y las Features, son el principal artefacto de requerimiento usado por los equipos ágiles.

- Visión general de User Story

Una User Story es una breve declaración o intento de ella que describe algo que el sistema necesita hacer para el usuario.

En Scrum, el PO frecuentemente escribe las user stories, con entradas de los clientes, partes interesadas y el equipo. Actualmente, cualquier miembro del equipo con el suficiente conocimiento del dominio puede escribir una user story, pero es levantada al PO para que acepte y priorice las potenciales stories en el product backlog.

Las user stories son una herramienta para definir el comportamiento de un sistema para que **sea entendible para los desarrolladores y los usuarios**.

Los detalles del comportamiento no aparecen en esta breve declaración, son dejados para ser desarrollados luego a través de conversaciones y criterios de aceptación entre el equipo y el PO.

- Las User Stories no son requerimientos

Son materialmente diferentes en ciertas formas, por ejemplo:

- Representan pequeños incrementos de funcionalidad valuada, que puede ser desarrollada en un período de días a semanas.
- Son relativamente fáciles de estimar, entonces el esfuerzo para implementar la funcionalidad puede ser rápidamente determinada.
- No se detallan al inicio del proyecto, sino que se elaboran justo a tiempo, evitando así una especificidad demasiado temprana, retrasos en el desarrollo, inventario de requisitos y un enunciado demasiado restrictivo de la solución.
- Las user stories y el código que se crea rápidamente después, sirven como entradas para la documentación, que también se desarrolla de forma incremental.

Forma de una User Story

- Carta, Conversación y Confirmación

Los tres elementos de una user story son:

- Carta: representa 2 o 3 oraciones usadas para describir la intención de una story. La carta sirve como ficha memorable, que resume la intención y representa un requerimiento más detallado, cuyos detalles aún están por determinarse. Representa una "promesa de una conversación" de la intención.
- Conversación: representa una discusión entre el equipo, el cliente, el PO y otros interesados, que es necesaria para determinar un comportamiento más detallado para implementar la intención.
- Confirmación: representa las condiciones de satisfacción que serán aplicadas para determinar si la story cumple la intención tal como los requerimientos más detallados.

- La voz de una User Story

En estos últimos años, una forma estandarizada fue aplicada que fortalece la construcción de una user story significativamente. La forma es la siguiente:

"*Como <rol>, yo puedo <actividad> par que <valor de negocio>"*

El "rol" permite una segmentación del producto funcionalidad y suele extraer otras necesidades basadas en roles y el contexto de la actividad. La "actividad" típicamente representa el requerimiento necesario por el rol. Y el "valor" comunica el porqué la actividad es necesaria, lo que frecuentemente lidera al equipo para encontrar posibles actividades alternativas que pueden proveer el mismo valor con menos esfuerzo.

- Criterios de aceptación

Los criterios de aceptación no son un test funcionales o test unitarios, en realidad son condiciones de satisfacción siendo colocados en el sistema. Los anterior mencionados van mucho más profundos en testear todas las corrientes de funcionamiento, corrientes de excepciones, condiciones límite, y funciones relacionadas con la story.

- INVEST en buenas User Stories

Los equipos ágiles pasan un significativo tiempo en descubrir, elaborar y entender las historias de usuario y escribiendo tests de aceptación. Esto es porque hay que entender bien el objetivo del código. Por lo tanto, invertir en buenas historias de usuario, aunque sea en el último momento de responsabilidad, es un esfuerzo que merece la pena para el equipo.

Los atributos de una buena story se pueden describir a partir del acrónimo INVEST:

- **I**ndependent (Independiente):
  Independencia significa que una story puede ser desarrollada, testeada y potencialmente enviada por sí sola. Por lo tanto, su valor también es independiente. Muchas historias tienen, naturalmente, una secuencia para la construcción de una funcionalidad, y, aún así cada pieza puede agregar valor independientemente entre sí.
  Sin embargo, muchas dependencias sin valor sean técnicas o funcionales, también encuentran la forma de entrar a los backlogs y tienen que ser encontradas y eliminadas. Por ejemplo:

  As an administrator, I can set the consumer's password security rules so that users are required to create and retain secure passwords, keeping the system secure.

  As a consumer, I am required to follow the password security rules set by the administrator so that I can maintain high security to my account.

  La forma correcta de escribirlas sería:

  As an Administrator, I can set the password expiration period so that users are forced to change their passwords periodically.

  As an Administrator, I can set the password strength characteristics so that users are required to create difficult to hack passwords.

  Ahora, cada historia puede valerse por sí misma, y pueden ser desarrolladas, testeadas y enviadas independientemente.
- **N**egotiable (Negociable):
  A diferencia de los requerimientos tradicionales, una user story no es un contrato para funciones específicas, sino un marcador de posición para que los requisitos sean discutidos, testeados y aceptados. Este proceso de negociación entre el empleador y el equipo reconoce la legitimidad y la prioridad de lo aporte la empresa, y permite el descubrimiento a través de la colaboración y la retroalimentación.
  Agile está fundada en el concepto de que la aproximación es más efectiva para solucionar problemas en un ambiente dinámico de colaboración.
  La ausencia de requisitos demasiado restrictivos y detallados mejora la capacidad de los equipos y las empresas para hacer concesiones de funcionalidad y plazos de entrega.
- **V**aluable (Valioso):
  Valor es el atributo más importante de INVEST, y toda user story debe proveer algo de valor al usuario. Los backlogs son priorizados por valor y el éxito empresarial o fracaso esta basado en el valor que los equipos pueden enviar. Esto se basa en lo que Wake dijo:
  "Piense en una historia completa como un pastel de varias capas, por ejemplo, una capa de red, una capa de persistencia, una capa lógica y una capa de presentación. Cuando dividimos una historia [horizontalmente],

estamos sirviendo sólo una parte de ese pastel. Queremos dar al cliente la esencia de todo el pastel, y la mejor manera es verticalmente a través de las capas. Los desarrolladores suelen inclinarse por trabajar en una sola capa a la vez (y hacerla "bien"); pero una capa completa de base de datos (por ejemplo) tiene poco valor para el cliente si no hay una capa de presentación.“

Crear stories valiosas requiere la reorientación de las estructuras de desglose funcional de un enfoque horizontal a uno vertical. Crear historias que atraviesen la arquitectura para poder presentar valor al usuario y recabar sus comentarios lo antes posible.

Aunque normalmente el valor se centra en el usuario que interactúa con el sistema, a veces es más apropiado centrarle en un representante del cliente o una parte interesada clave.

- **E**stimable

  Mientras que una story de cualquier tamaño puede estar en el backlog, en orden de ser desarrollada y testeada en una iteración, el equipo puede estar apto para proveer una estimación aproximada de su complejidad y la cantidad de trabajo requerida para completarla.

  Si el equipo es incapaz de estimar una story, generalmente indica que la story es muy larga o incierta. Si es muy complejo de estimar, puede ser dividida en pequeñas stories. Si la historia es muy incierta para estimar, una spike técnica o funcional puede ser usada para reducir esta incertidumbre.
- **S**mall (Pequeña):

  Deben ser lo suficientemente pequeñas para ser completadas en una iteración, de otra forma no puede proveer ningún valor o ser considerada hecha en ese punto.
- **T**esteable:

  Si la story no parece ser testeable, entonces la story esta probablemente mal formada, compleja o depende de otras historias del backlog.

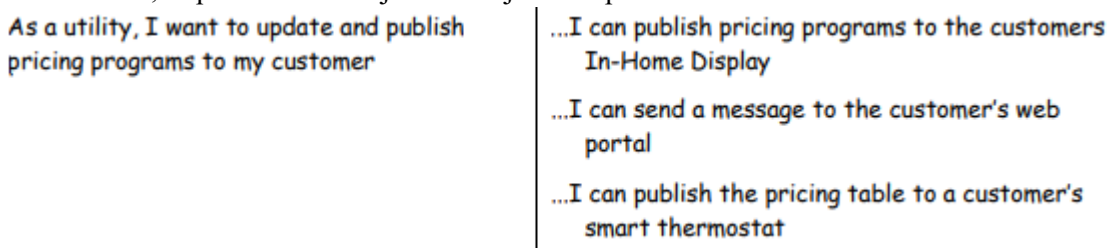
  En orden para asegurar la testeabilidad, las user stories comparten algunos errores comunes de testeabilidad con requisitos imprecisos. Palabras como “rápidamente”, “lindo”, “limpio”, etc. son fáciles de escribir, pero difíciles para testear porque significan cosas diferentes para diferentes personas, por lo que deben ser esquivadas. Y mientras estas palabras provean negociación, entendimiento, límites objetivos; ayudarán al equipo a la empresa compartir expectaciones sobre la salida y evitar grandes sorpresas.

Dividiendo User Stories

Existen 10 patrones comunes para dividir una story:

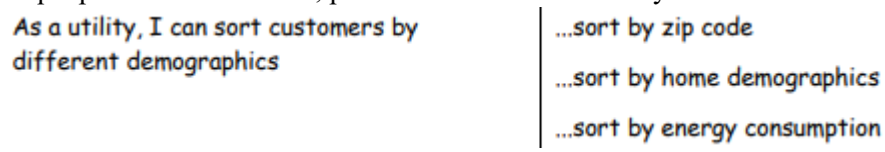
1) Pasos de flujo de trabajo:

   Identificar los pasos específicos que un usuario realiza para llevar a cabo un flujo de trabajo específico, y, a continuación, implementar el flujo de trabajo en etapas incrementales.

2) Variaciones de las reglas de negocio:

   Algunas stories parecen sencillas. Sin embargo, a veces las reglas de negocio son más complejas o extensas de lo que parece. En este caso, puede ser útil dividir la story en varias.

3) Mayor esfuerzo:

   A veces una story puede ser dividida en muchas partes donde el mayor esfuerzo va a ir en la implementación de la primera. En el siguiente ejemplo, la infraestructura de procesamiento debe construirse para soportar la primera story, después añadir más funcionalidad debería ser relativamente trivial.

As a user, I want to be able to select/change my pricing program with my utility through my web portal	...I want to use Time-of-Use pricing ...I want to Pre-pay for my energy ...I want to enroll in Critical-Peak-Pricing

4) Simple/Complejo:
En una discusión preguntar "¿qué es lo más simple que puede funcionar?". Capturar esa simple versión como una story en sí, y luego que las variantes y complejidades salgan con su propia story.

As a user, I basically want a fixed price, but I also want to be notified of Critical-Peak-Pricing events.	...respond to the time and the duration of the critical peak pricing event ...respond to emergency events

5) Variaciones en los datos:
Las variaciones de los datos son otro recurso para vigilar y complejo. Considerar agregar stories justo a tiempo después de la versión más simple.

As a utility, I can send messages to customers	...in English. ...in Spanish ...in Arabic., etc.

6) Métodos de entrada de datos:
A veces, la complejidad es la interfaz de usuario. En ese caso, dividir la story para construir la UI más simple y después complejizarla después.

As a user, I can view my energy consumption in various graphs	...using bar charts that compare weekly consumption ...in a comparison chart, so I can compare my usage to those who have the same or similar household demographics

7) Aplazar las cualidades del sistema:
A veces, la implementación inicial no es tan difícil y la mayor parte del esfuerzo consiste en hacerla más rápida, o fiable, o precisa o escalable. Sin embargo, el equipo puede aprender mucho de la implementación de base, y debe tener algún valor para un usuario que, en de otro modo, no podría hacerlo todo.

As a user, I want to see real-time consumption from my meter	...interpolate data from the last known reading ...display real time data from the meter

8) Operaciones (CRUD):
Las palabras "gestionar" o "controlar" indican que la story abarca varias operaciones, lo que puede ofrecer una forma natural de dividir la story.

As a user, I can manage my account.	...I can sign up for an account. ...I can edit my account settings. ...I can cancel my account. ...I can add more devices to my account

9) Escenarios de Casos de Uso:
Si el caso de uso fue desarrollado par representar complejidad, entonces la story puede frecuentemente dividirse de acuerdo con los diferentes escenarios.

"I want to enroll in the energy savings program through a retail distributor."	Use Case/Story #1 (happy path): Notify utility that consumer has equipment Use Case/Story #2: Utility provisions equipment and data, notifies consumer Use Case/Story #3 (alternate scenario): Handle data validation errors

10) Romper una spike:
En algunos casos, una story puede ser demasiado grande o compleja, o tal vez esta mal la implementación. En ese caso, construir una spike para resolverlo y, a continuación, dividir la story en función de los resultados.

Spikes

Spikes son un tipo especial de story que es usada para despejar el riesgo e incertidumbre en una user story u otra faceta del proyecto. Pueden ser usadas por muchas razones:

- El equipo puede no tener conocimiento del nuevo dominio, y las Spikes pueden ser usadas para búsqueda básica para familiarizarse con la nueva tecnología o domino.
- La story puede ser muy grande para ser estimada apropiadamente, y el equipo puede usar una spike para analizar el comportamiento, por lo que ellos pueden dividir la story en piezas estimables.
- La story puede contener un riesgo técnico significante, y el equipo debería hacer algo de búsqueda o prototipado para ganar confianza en la tecnología para permitirles agregar la user story en otros timebox.
- La story puede contener un riesgo funcional significante, mientras que el intento de entender la story, no está claro cómo el sistema necesita interactuar con el usuario para conseguir el beneficio implicado.

- Spikes técnicas y funcionales

Las spikes técnicas son usadas para investigar varias aproximaciones técnicas en el dominio de la solución. Por ejemplo, puede ser utilizada para determinar una decisión de construcción o compra, evaluación de una potencial performance o impacto de una nueva user story, etc.

Las spikes funcionales son usadas cuando hay una incertidumbre significativa sobre cómo el usuario debería interactuar con el sistema. Suelen evaluarse mejor mediante algún nivel de creación de prototipos, ya sean maquetas de la interfaz de usuario, esquemas, flujos de páginas, o cualquier otra técnica que resulte más adecuada para obtener una retroalimentación de los clientes o las partes interesadas.

A veces, algunas user stories necesitan ambos tipos de spikes:

> As a consumer, I want to see my daily energy use in a histogram, so that I can quickly understand my past, current, and likely near term, future energy consumption.

In this case, a team might create two spikes:

> Technical Spike: Research how long it takes to update a customer display to current usage, determining communication requirements, bandwidth, and whether to push or pull the data.
>
> Functional Spike: Prototype a histogram in the web portal and get some user feedback on presentation size, style, and charting attributes.

Lineamientos para las Spikes

- Estimable, Demostrable y Aceptable

Como otras stories, las spikes son puestas en el backlog, estimadas para encajar en una iteración. Los resultados de las spikes son diferentes a los de las stories, generalmente producen información, no código. La spike debería desarrollar solo la información suficiente para resolver la incertidumbre.

La salida de una spike es demostrable para el equipo. Esto brinda visibilidad a la investigación y los esfuerzos arquitectónicos y también ayuda a construir una pertenencia colectiva y compartir responsabilidad para las decisiones claves que se van tomando.

Y como las otras stories, las spikes son aceptadas por el PO cuando los criterios de aceptación fueron cumplidos.

- La excepción, no la regla

Toda user story tiene incertidumbre y riesgo. El equipo descubre la mejor solución a través de discusiones, colaboraciones, experimentaciones y negociación. Por ello, en cierto sentido, toda user story contiene alguna actividad de spike. La meta de un equipo agile es aprender cómo abrazar y efectivamente corregir esa incertidumbre en cada iteración. Una spike, puede ser reservada para desconocidos más críticos y largos.

- Considerar implementar la spike en un sprint diferente a las stories resultantes

Planear que la spike y el resultado estén en la misma iteración es riesgoso, y debería ser evitado. De todos modos, si la spike es pequeña y pareciera que puede ser encontrada una rápida solución, no hay nada malo en completar la story resultante en la misma iteración.

Libro: *"User Stories Applied for Agile Software"*

Capítulo 1: Trabajo Preliminar

- ¿Qué es una story?

Una user story es una pieza de funcionalidad relativamente pequeña que será valiosa en el software del usuario. Son tradicionalmente escritas en notas ya que son de poca elegancia tecnológica.

- ¿Dónde están los detalles?

Hay una sola cosa por decir "Un usuario puede hacer una reservación de hotel". Otra cosa es codificar y probar un programa con este requisito.

Muchos de los detalles pueden ser expresados en stories adicionales. Cuando una story es muy grande, se la llama épica, ésta puede ser dividida en dos o más stories de menor tamaño. De todas formas, no se va a seguir dividiendo las stories hasta cubrir cada uno de todos los detalles. Similarmente, la user story no necesita ser argumentada en un estilo típico de documentación de requerimientos.

Mejor que escribir todos estos detalles como stories, es enfocarse en la discusión entre el equipo de desarrollo y el cliente. La conversación es la clave, no la nota de la story. Las stories no son obligaciones contractuales.

- ¿Qué tan larga debe ser?

Las descripciones de tests (criterios de aceptación) deben ser cortas e incompletas. Los tests pueden ser agregados o eliminados en cualquier momento. El cometido es agregar información adicional sobre la story para que los desarrolladores sepan cuando hayan terminado

- Carta, Conversación y Confirmación

Ron Jeffries ha dado la maravillosa aliteración de que las stories tienen tres aspectos críticos: Carta, Conversación y Confirmación. Esto significa que la user story está representada por la carta que está escrita, la conversación entre el cliente y los desarrolladores que definen y refinan el significado de la story y finalmente por las pruebas de aceptación que confirman que la story fue correctamente programada. Mientras que la carta contiene el texto de la story, los detalles son encontrados en la conversación y la confirmación.

Capítulo 2: Qué no son las Stories

- Las User Stories no son IEEE 830

La Computer Society of the Institute of Electrical and Elecronics Engineers (IEEE) publicó un grupo de lineamientos sobre cómo escribir especificaciones de requerimientos.

La característica más destacada del estilo IEEE 830 para la especificación de requerimientos es el uso de la frase "El sistema debería…" la cual es la forma recomendada para escribir requerimientos.

Documentar los requerimientos del sistema a este nivel es tedioso, propenso al error y consume mucho tiempo. Adicionalmente, un documento de requerimientos escrito de esta manera, francamente, es aburrido de leer.

Los requisitos del IEEE 830 han descarriado muchos proyectos porque centran la atención en una lista de requisitos en lugar de los objetivos del usuario. Centrándose en lo que quiere el usuario para el nuevo producto, es mejor que una lista de atributos. Estamos preparados para diseñar una mejor solución para lo que necesita el usuario.

- Las User Stories no son Casos de Uso

Un caso de uso es una descripción generalizada de una interacción entre el sistema y un actor, donde el actor es un usuario u otro sistema. Los casos de uso pueden ser escritos de manera desestructurada o conforme a una plantilla.

Una de las más obvias diferencias entre las stories y los casos de uso son su alcance. Un caso de uso es casi siempre mucho más largo que una story. Esto nos lleva a la observación de que una user story es similar a un escenario del caso de uso. Cada story no necesariamente es equivalente al escenario exitoso principal. Además, se diferencian en el nivel de completitud.

Otra importante diferencia entre ellas es su longevidad. Los casos de uso son usualmente artefactos permanentes que continúan existiendo mientras el producto esta en desarrollo activo o mantenimiento. Las stories, de otro lado, no se pretenden que sobrevivan a la iteración en la cual fueron agregadas al software.

También los casos de uso son más propensos a incluir detalles de la interfaz de usuario.

Finalmente, los casos de uso y las stories son escritas con diferentes propósitos. Los casos de uso son escritos en un formato aceptable para los clientes y los desarrolladores para que cada uno pueda leer y estar de acuerdo con ellos. Su propósito es documentar un acuerdo entre el cliente y el equipo de desarrollo. Las stories, son escritas para facilitar la planeación de la iteración y reléase y para servir como marcador para conversaciones que descubren las necesidades detalladas del usuario.

- Las User Stories no son escenarios

Un escenario es una descripción detallada de la interacción entre el usuario con la computadora. Los escenarios de diseño de interacción no son lo mismo que un escenario de un caso de uso. De hecho, un escenario de diseño de interacción es típicamente más largo, o más abarcador, que incluso un caso de uso. La principal diferencia entre estos y las user stories son su enfoque y su detalle.

Capítulo 6: Escribiendo Stories

- Formato

Una de las principales ventajas te usar cartas sobre software es que su naturaleza de baja tecnología es un constante recordatorio que las stories son imprecisas. Una práctica común en usar cartas es escribir los criterios de aceptación detrás de la carta.

- “Las stories son promesas de conversación”

Porque las stories son escritas para recordar que hay que conversar, las stories no necesitan incluir todos los detalles relevantes. El desafío viene en aprender incluir el suficiente detalle.

Las notas en la carta permiten al desarrollador y al cliente resumir la conversación que tuvieron antes. Idealmente, la conversación puede reanudarse fácilmente independientemente de que el mismo desarrollador haya participado o no en las conversaciones previas. De todas formas, en muchos casos especificar detalles muy temprano crea más trabajo.

Si pensamos a cerca que la story es un recordatorio para el desarrollador y el cliente para tener una conversación, entonces es útil pensar que la story es un contenedor de:

- una frase o dos que actúa como recordatorio de mantener una conversación
- Notas sobre problemas a resolver durante la conversación

Los detalles que fueron ya determinados a través de conversaciones pasan a ser tests.

Libro: *"No silver bullets"*

Divido en "esencia" las dificultades inherentes a la naturaleza del software, y "accidentes" a las dificultades que actualmente asisten a su producción pero que no son inherentes. Creo que la parte más difícil en la creación de un software es la especificación, diseño y prueba de la construcción de esta base conceptual, no el trabajo de representar y probar la calidad de la representación.

Si esto es cierto, crear un software será siempre difícil. No hay intrínsecamente ninguna bala de plata. Consideremos las propiedades inherentes de esta esencia irreducible de los sistemas modernos de software:

- Complejidad: un aumento de una entidad software no es simplemente una repetición de los mismos elementos a una mayor escala, es necesario un aumento en el número de distintos elementos. En la mayoría de los casos, los elementos interactúan entre si de una forma no lineal, y la complejidad del todo incrementa mucho más que linealmente.
  La complejidad del software es una propiedad esencial, no accidental. Por lo tanto, las descripciones de una identidad software que dejan de lado su complejidad también lo hacen de su esencia.
  Muchos de los típicos problemas de desarrollar productos software provienen de esta complejidad esencial y su no-linealidad incrementa con el tamaño.
- Conformidad: gran parte de la complejidad que él debe dominar es complejidad arbitraria, forzada sin ton ni son por las muchas instituciones humanas y sistemas a los cuales sus interfaces deben conformar.
- Mutabilidad: la entidad software está constantemente sujeta a presiones de cambio. Esto es así porque el software de un sistema abarca su función, y la función es la parte que más siente la presión de cambio. Y esto es en parte porque el software puede ser cambiado con mayor facilidad. Todo software exitoso es cambiado. Se trabajan dos procesos.
  Primero, cuando un producto software es útil, las personas lo prueban llevándolo al límite o más allá de su dominio original. La presión para extender las funciones viene principalmente de usuarios a los que les gustan las funciones básicas y les inventan nuevos usos.
  Segundo, un software exitoso sobrevive más allá de la vida normal de máquina para la que fue creada.
- Invisibilidad: el software es invisible, las abstracciones geométricas son herramientas poderosas. Tanto como intentamos esquematizar la estructura del software, nos encontramos con que constituye no uno, sino varios gráficos generales superpuestos uno sobre el otro. Los varios gráficos pueden representar el flujo de control, el flujo de datos, los patrones de dependencia, la secuencia de tiempo y las relaciones nombre-espacio.
  Esta carencia no solo le impide el proceso de diseño a una sola mente, sino que también dificulta gravemente la comunicación entre mentes.

Últimos avances que resolvieron dificultades accidentales

Si analizamos los tres pasos del desarrollo de la tecnología software que han sido más fructíferos en el pasado, descubriremos que cada uno atacó a una de las principales dificultades accidentales en la construcción de software.

- Lenguaje de alto nivel: libera a un programa de gran parte de su complejidad accidental. En la medida en que el alto nivel de lenguaje incorpora las construcciones que uno quiera en el programa abstracto y evite todos los de menor nivel, éste elimina todo un nivel de complejidad que nunca fue inherente al programa en absoluto.
- Tiempo compartido (multiprogramación): significó una gran mejora en la productividad de los programadores y en la calidad de sus productos. El tiempo compartido ataca una dificultad bastante diferente. Ese preserva la inmediatez, y por lo tanto permite que se mantenga una visión general de complejidad. Una lenta rotación, como las complejidades lenguaje-máquina, es una dificultad accidental del proceso de software. Los límites de la contribución potencial de tiempo compartido se derivan directamente.
- Entornos de programación unificados: atacan dificultades accidentales que derivan de la utilización de programas individuales juntos, mediante el suministro de bibliotecas integradas, archivos de formato unificado y de tuberías y filtros. Como resultado de esto, las estructuras conceptuales que básicamente podrían siempre llamar, suministrar datos y usarse entre sí pueden, de hecho, hacerlo más fácil en la práctica.

Esperanzas de la plata

Ahora consideremos los desarrollos técnicos que son considerados como posibles "balas de plata" con más frecuencia:

- Programación orientada a objetos:

Existen los tipos de dato abstractos y jerárquicos. Los abstractos son un tipo de objeto que debe ser definido con un nombre, un conjunto de valores y un conjunto adecuado de operaciones y no por su estructura de almacenamiento, la cual debe ser ocultada. Los jerárquicos le permite a uno definir interfaces generales que pueden ser mejoradas más tarde, administrándoles los tipos de subordinación.

Los dos conceptos son diagonales; uno puede estar jerarquizado o disimulo sin jerarquización. Ambos conceptos representan verdaderos avances en el arte de crear software.

Cada uno remueve otra dificultad accidental del proceso, permitiéndole al diseñador expresar la esencia del diseño sin tener que expresar una gran cantidad de material sintáctico cuyo contenido no añade información. No obstante, estos avances no pueden hacer más que eliminar las dificultades accidentales de la expresión de diseño.

- Inteligencia artificial

Muchas personas esperan el desarrollo en la inteligencia artificial que proporcione el avance revolucionario que significará enromes ganancias en cuanto a productividad y calidad del software. Para saber por qué debemos analizar lo que se entiende por "inteligencia artificial".

Dos definiciones muy distintas de IA son comúnmente usadas hoy en día:

1. El uso de computadores para resolver problemas que anteriormente sólo podían ser resueltos mediante la aplicación de inteligencia humana.
2. El uso de un conjunto específico de técnicas de programación conocida como heurística o programación basada en las normas. En este enfoque se utilizan humanos expertos para determinar las heurísticas o reglas básicas que usan para resolver problemas. El programa es diseñado para resolver un problema de la forma en que los seres humanos parecen hacerlo.

Un sistema experto (según la definición 2) es un programa que contiene un motor de inferencia generalizado y una norma de base, toma de entrada de datos y asunciones, explora las interferencias que derivan de la norma de base, procura conclusiones y consejos, y ofrece explicar los resultados mostrando su razonamiento al usuario. Tales sistemas ofrecen ventajas claras sobre los algoritmos diseñados para llegar a las mismas conclusiones de los mismos problemas:

- La tecnología motor de la inferencia es desarrollada en una forma independiente de aplicaciones, y luego se aplica a muchos usos. Uno puede justificar un gran esfuerzo en la inferencia de los motores.
- Las partes variables de los materiales característicos de la aplicación están codificados en la norma de base de forma uniforme y se proporcionan herramientas para el desarrollo, la evolución, la prueba y la documentación de la norma de base.

El poder de tales sistemas no proviene de mecanismos de inferencia lujosos, sino más bien de mecanismos con bases de conocimiento cada vez más sustanciosas que reflejan de manera más precisa el mundo real.

A quien desarrolla un programa se le presentan muchas dificultades en el camino por una pronta realización de un útil sistema consejero experto. Una parte fundamental de nuestro imaginario escenario es el desarrollo de formas fáciles de obtener desde la especificación de programas-estructura a la generación automática o semiautomática de las reglas de diagnóstico. Aún más difícil e importante la doble tarea de adquisición de conocimientos: la búsqueda de expertos elocuentes, auto analíticos, que conozcan la razón por la que hacen las cosas, desarrollando técnicas eficientes para la extracción de lo que saben y separando en sus componentes las bases de normas. El prerrequisito esencial para la construcción de un sistema experto es disponer de un experto.

La contribución más poderos de los sistemas expertos, sin duda será poner al servicio de programadores novatos la experiencia y la sabiduría acumulada de los mejores programadores.

- Programación automática

Por casi 40 años la gente ha estado anticipando y escribiendo acerca de esto o sobre la generación de programas resolviendo problemas del orden de la especificación de problemas.

Parmas sugiere:

"*En resumen, la PA siempre ha sido un eufemismo de la programación con un mayor nivel de lenguaje disponible actualmente para el programador."*

Él argumenta, en esencia, que en la mayoría de los casos tiene que ser dada la explicación del método de solución y no del problema en sí.

- Gráfica de la programación

Un tema favorito para la tesis de doctorado en ingeniería de software es programación gráfica o visual, la aplicación de desde gráficos de computador a diseño de software. Algunas veces la promesa mantenida por ese enfoque es postulada como una analogía con diseño de circuito integrado VLSI, en el cual gráficos computacionales desempeñan un papel tan provechoso. A veces el teórico justifica la metodología teniendo en cuenta los diagramas de flujo como el ideal programa-diseño y suministrando poderosos planteles para construirlos.

La tecnología hardware tendrá que desarrollarse considerablemente antes de que el alcance de nuestras extensiones sea suficiente para el diseño de escritorio de software.

Esencialmente, como he argumentado antes, el software es muy difícil de visualizar. Ya sea por los diagramas de control de flujo, referencias variables cruzadas, flujo de datos, estructuras jerárquicas de datos, o lo que sea, uno siente tan solo una dimensión del intrincadamente elaborado software. Si uno fuerza todos los diagramas generados por las muchas visiones relevantes resultad difícil extraer una visión general.

- Verificación de programa:

¿Se podrá encontrar tal vez una bala de plata luego de eliminar los errores en el origen o en la fase de diseño del sistema? ¿Pueden la productividad y la fiabilidad del producto ser radicalmente mejorados, siguiendo así la estrategia tan opuesta de proveer diseños sin errores, antes de derrochar el inmenso esfuerzo en implementarlos y probarlos?

Esto es un concepto muy poderoso y será muy importante para cosas como un seguro manejo de núcleos de sistema. Sin embargo, esta tecnología no promete ahorrar mano de obra. Las comprobaciones significan tanto trabaja que tan solo unos pocos programas importantes han sido verificados alguna vez. Aunque la verificación podría reducir la carga de lo que significa probar los programas, no la puede eliminar.

- Entornos y herramientas:

¿Cuántas ganancias se pueden esperar de la explotación de investigaciones dirigidas hacia mejores entornos de programación? Los editores inteligentes con lenguaje específico son avances que aun no se utilizan masivamente en la práctica, pero lo máximo que pueden prometer es no tener errores sintácticos ni errores semánticos simples.

Tal vez el mayor logro de los entornos de programación será el uso de bases de datos integradas para seguirles el rastro al gran número de detalles que deben ser recordados con precisión por el programador, y mantenido al día por un grupo de colaboradores en un solo sistema. Pero, por su propia naturaleza los beneficios serán insignificantes.

- Estaciones de trabajo:

¿Qué ganancias puede esperar el material gráfico de software del incremento seguro y veloz de la capacidad de memoria y poder de estación de trabajo individual? Ciertamente le damos la bienvenida a estaciones de trabajo más poderosas. No podemos esperar mejoras mágicas de ellas.

Ataques prometedores en la esencia conceptual

Si como se cree, los componentes conceptuales de la tarea están tomando la mayor parte del tiempo, entonces ninguna cantidad de actividad en los componentes de la tarea, que son solamente la expresión de conceptos, puede brindar ganancias considerables.

Por lo tanto, debemos considerar esos ataques dirigidos a la esencia del problema del software, la formulación de esas complejas estructuras conceptuales. Afortunadamente algunos de estos ataques son prometedores:

- Comprar vs crear

La solución más radical y posible para la construcción de software es no construirlo. Cada día esto es más fácil, más y más vendedores ofrecen más y mejores productos software para una variedad de aplicaciones. Mientras que nosotros, ingenieros de software, hemos trabajado en la metodología de producción, la revolución de los computadores personales ha creado no uno, sino muchos mercados masivos para software.

Cualquiera de estos productos es más barato que construir uno nuevo. Tales productos tienden a ser mucho mejor certificados y de alguna forma mejor mantenidos que los de creación personal. El desarrollo de los mercados masivos es, según yo, la más profunda tendencia en ingeniería de software.

El gran cambio ha sido en el porcentaje de costos de hardware y software. En 1960 el comprador de una máquina de dos millones de dólares sentía que podía costear $250.000 más por un programa de planilla personalizado, uno que se escabulla fácilmente en el hostil ambiente social computacional. Hoy en día, el comprador de una máquina de oficina de $50.000 no puede costear un programa de planilla personalizado, así que adapta el procedimiento de planilla a los paquetes disponibles. Los computadores son tan comunes actualmente que las adaptaciones son acogidas rápidamente.

Muchos usuarios operan ahora sus computadores a diario en varias aplicaciones sin siquiera escribir un programa. De hecho, muchos de esos usuarios no pueden escribir nuevos programas para sus máquinas, pero sin embargo han logrado resolver nuevos problemas con ellos.

Requerimientos, refinamiento y rápida creación de prototipos

La parte más difícil en la creación de un software es decidir precisamente qué construir. Ninguna otra parte del trabajo conceptual es tan difícil como establecer los detallados requerimientos técnicos, incluyendo todas las interfaces a las personas, máquinas y otros sistemas de software. Ninguna otra parte del trabajo paraliza los resultados del trabajo si se hace mal. Ninguna otra parte es más difícil de arreglar después.

Por lo tanto, la función más importante que el creador de software desempeña para el cliente es la extracción iterativa y refinamiento de los requerimientos del producto. Así es que para plantear cualquier diseño software es necesario permitir una extensa iteración entre el cliente y el diseñador como parte de la definición del sistema.

Iría más lejos y sostendría que es realmente imposible para un cliente, incluso que trabaje con ingeniería de software, especificar de manera completa, precisa y correcta los exactos requerimientos de un producto software moderno antes de probar algunas versiones del producto.

Uno de los esfuerzos tecnológicos más prometedores que ataca la esencia y no los accidentes del problema software es el desarrollo de aproximaciones y herramientas para la rápida creación de prototipos de sistemas. El propósito del prototipo es hacer real la estructura conceptual especificada, para que el cliente pueda probar su consistencia y utilidad.

Grandes diseñadores

Estudio tras estudio muestra que los mejores diseñadores producen estructuras que son más rápidas, más pequeñas, más simples, más pulcras y son producidas con menor esfuerzo. Las diferencias entre el gran diseñador y el promedio son enormes.

Creo que el esfuerzo más importante que podemos alcanzar es el desarrollo de métodos para aumentar grandes diseñadores. Ninguna organización software puede ignorar este desafío. Buenos gerentes no son tan escasos como los buenos diseñadores. Grandes gerentes y diseñadores son muy escasos. La mayoría de las organizaciones hace considerables esfuerzos por encontrar los prospectos administrativos; no conozco ninguna que haga el mismo esfuerzo para encontrar grandes diseñadores de los que la excelencia de sus productos depende.

¿Cómo aumentar a los grandes diseñadores?

- Identificar sistemáticamente a los mejores diseñadores lo más pronto posible.
- Asignar un orientador profesional para que se haga responsable del prospecto.
- Desarrollar y mantener un plan profesional de desarrollo para cada prospecto.
- Dar oportunidades de crecimiento a los diseñadores, para que interactúen y se estimulen entre sí.