TESIS DE MAESTRIA

Un Enfoque para Mejorar la Productividad en Empresas de Desarrollo de Software

por

Gustavo Andrés López

Licenciado en Sistemas de Información
1998 Universidad del Salvador

Presentado a la Escuela de Posgrado del ITBA y de la EOI de España
en cumplimiento parcial
de los requerimientos para la obtención del título de

Magister en Dirección Estratégica y Tecnológica (Argentina)
Master Executive en Dirección Estratégica y Tecnológica (España)

En el Instituto Tecnológico de Buenos Aires

Junio de 2015

Firma del Autor _____

Instituto Tecnológico de Buenos Aires
Fecha: 4 de Junio de 2015

Certificado por _____

Lic. Santiago Ceria, Profesor de Sistemas de Información
Instituto Tecnológico de Buenos Aires
Tutor de la Tesis

Aceptado por _____

MSc. Ing. Diego Luzuriaga, Director del Programa
Instituto Tecnológico de Buenos Aires

Miembros del Jurado:

Agradecimientos

Va mi profundo agradecimiento a mis colegas de la industria del software que a través de encuestas, entrevistas, o simples charlas brindaron su experiencia y opinión como aporte a este trabajo. Al ITBA y a sus docentes por el conocimiento que ponen diariamente a disposición de sus alumnos. A mi tutor por las horas dedicadas y consejos sobre el enfoque de mi investigación. Y muy especialmente a mi esposa y mis 3 hijos que con mucho esfuerzo, pero con mucho amor, me dieron el espacio y el tiempo necesarios para terminar este trabajo, que no fue poco.

Dedicatoria:

Este trabajo está dedicado a todos aquellos que tienen por delante el desafío de producir software de calidad en tiempos cortos y costos competitivos, esperando aportar algo de claridad, algunas ideas y lineamientos que los ayuden a lograr su cometido

Índice de Capítulos:

Índice de Cuadros:

Índice de Figuras:

Abstract

Luego de más de 20 años de experiencia en la industria del software, se ven ya con cierta normalidad que los proyectos de desarrollo fallan sistemáticamente en lograr los resultados esperados dentro de los tiempos y costos previstos desde un principio, y con niveles de calidad adecuados.

Es normal ver que las etapas previstas para la construcción y entrega de un sistema toman más tiempo e insumen más costo del esperado; y además, en muchos casos, siguen a esto varios meses, e incluso años, en los que es necesario seguir trabajando para mantener el sistema funcionando, realizando ajustes o solucionando fallas que pasaron de manera inadvertida hasta el día en que alguna condición particular las hace aparecer. Aparece entonces un interrogante que no parece fácil de responder ¿Por qué es tan difícil y complejo desarrollar software que funcione bien desde el primer día y satisfaga las necesidades de los clientes? Hay una relación compleja entre la funcionalidad de los sistemas construidos y la cantidad de trabajo necesario para hacerlo, y parece ser esta una relación que podría ser ampliamente mejorada.

Bajo el concepto de la productividad, este trabajo pretende estudiar el porqué de este fenómeno y qué se puede hacer para lograr una relación más favorable para el negocio de los que se dedican a producir software. Desde una definición de Productividad general y otra aplicada a la industria del software, en este trabajo desarrolla ideas, definiciones y sugerencias de cómo medir y cómo mejorar la productividad, se estudian los factores que la afectan y cómo las diferentes metodologías que se usan hoy tratan el tema.

El trabajo finaliza en un enfoque, una propuesta, una posible forma de encaminar la actividad de desarrollo de software en la que se intenta contemplar y aprovechar todas las prácticas y enfoques que a lo largo de los años la industria del software, sus autores e investigadores más importantes han identificado como las que pueden ayudar a obtener la productividad deseada.

Introducción

Relevancia:

En un ambiente de negocios cada vez más dinámico y exigente, resultan ganadores aquellos que puedan generar ventajas competitivas claras que los posicione siempre un paso delante de sus competidores. En la industria del desarrollo de software, estas ventajas podrían traducirse en las capacidades que tenga una empresa para desarrollar software innovador y de buena calidad, que realmente satisfaga las necesidades de negocio de sus clientes, dentro de tiempos y costos lo más bajos posibles.

Esto hace pensar que se trata en parte de maximizar la eficiencia de la actividad de construir software para poder entregar más rápido, habiendo invertido la menor cantidad posible de materia prima y con un nivel de calidad que haga que el proyecto termine cuando debe terminar y no se prolonguen en forma indefinida las actividades de ajuste y puesta a punto.

Empresa que pueda hacer esto, logrará atraer y retener a sus clientes a la vez que mejorará sus márgenes de ganancia; porque logrará producir en forma más económica algo que cumpla las expectativas de sus clientes. Se trata simplemente de esto, la capacidad de sobrevivir y crecer en un entorno dinámico y muy competitivo, cosa que podrán hacer solo aquellos que logren buena productividad.

Siendo que Argentina es un país que busca posicionarse en el mundo como generador y exportador de software, es muy importante que sus empresas pongan foco en generar estas ventajas competitivas, lo que ayudará en definitiva no solo al crecimiento de las organizaciones en forma individual sino también al crecimiento de la industria nacional y al crecimiento de nuestra economía.

Definición y alcance del problema:

El foco de este trabajo está puesto en entender el problema de la productividad en desarrollo de software, y dar líneas o pautas para su medición y mejora que puedan ser de utilidad para cualquier empresa que se dedique a este negocio.

Se recorrerán y analizarán definiciones de productividad, formas de medirla específicas de la industria del software, se intentará identificar cuáles son los factores más importantes que afectan la productividad en esta actividad y se analizarán trabajos de investigación anteriores y metodologías en uso hoy para entender cómo cada una enfoca el problema de la productividad y propone mejorarlo.

Estado del conocimiento:

Mucho se ha escrito sobre desarrollo de software y métodos para llevar adelante la actividad de forma eficiente, pero la realidad es que la industria del software es una industria relativamente nueva e inmadura que sufre cierta inestabilidad de sus métodos y procedimientos porque estos cambian al ritmo que lo hace la tecnología que sustenta la actividad.

Desde los primeros métodos que intentaron darle al desarrollo de software un enfoque ingenieril, con procedimientos y métodos formales, y etapas de producción inspiradas en los procesos industriales de la época, hasta los métodos actuales que reconocen más la naturaleza humana de la actividad donde la creatividad y el talento juegan un papel importante, han pasado una gran cantidad de expertos, estudios, libros y trabajos de todo tipo tratando de echar luz sobre el tema.

La realidad es que, a pesar de lo estudiado, dicho y escrito, todavía hoy los proyectos sufren los mismos problemas de los primeros años: atrasos, fallas, sobre-costos, insatisfacción de los clientes; señal de que todavía hay trabajo por hacer.

Hipótesis:

La productividad general en actividades de desarrollo de software no está en su nivel óptimo y aún puede ser mejorada. Esta es la hipótesis que este trabajo intentará demostrar, aclarar algunos conceptos y presentar un enfoque que permita lograr una mejora en este aspecto.

Metodología

Partiendo de la definición general de productividad, y de cómo esta definición puede ser aplicada al software con sus mediciones y datos particulares, se comenzó a investigar el tema a partir de 2 libros que fueron tomados como base. Luego, lecturas adicionales, la búsqueda intensiva de material en internet, y otras publicaciones referidas por los especialistas consultados, fueron marcando el camino de esta investigación hasta el enfoque propuesto y la conclusión presentada.

No hay una demostración matemática, estadística o científica que lleve a concluir que la productividad puede ser mejorada. Es la propia experiencia profesional del autor, la de los colegas consultados y la bibliografía disponible las que dan cuenta de esta realidad. El desafío es dar un paso más hacia entender cómo hacerlo.

Revisión bibliográfica:
Se identifica en los Anexos toda la bibliografía consultada, incluyendo libros, informes, papers y otros materiales publicados en internet.

Objetivos de la investigación:
El objetivo de esta investigación es profundizar en el conocimiento y entendimiento de lo que significa la productividad en actividades de desarrollo de software, cómo podemos medirla teniendo en cuenta la complejidad adicional que representa la intangibilidad del software, y estudiar los factores que la afectan en forma positiva y negativa. Estudiar

también los métodos más conocidos y usados en la industria para entender cómo cada uno de ellos da tratamiento a estos factores y con cuales prácticas abordan el tema de la productividad.

Luego, presentar un enfoque ordenado, una idea de cómo encaminar un proyecto de software, o qué hay que tener en cuenta para mejorar la productividad de una empresa o equipo, teniendo en cuenta todos estos factores que fueron descubiertos.

Interrogantes de investigación:

- ¿Qué es la productividad y cómo podemos medirla en software?
- ¿Por qué los proyectos de software presentan todavía problemas de productividad?
- ¿Qué relación tiene la productividad con la calidad?
- ¿Qué relación tiene la productividad con el éxito de los negocios de software?
- ¿Cómo enfocan los métodos actuales el tema de la productividad?
- ¿Cuáles son las buenas prácticas que cada uno toma como base del método?
- ¿Cuáles son los factores más importantes?
- ¿Qué técnicas o buenas prácticas tienen en cuenta estos factores y de qué manera?
- ¿En definitiva, qué hay que tener en cuenta para mejorar la productividad?

Limitaciones y restricciones:

A pesar de que la actividad de desarrollo de software es bastante standard y similar a nivel internacional porque las tecnologías, técnicas y métodos en uso trascienden rápidamente las fronteras, se ha considerado como espacio de realización y aplicación de este estudio a las empresas PYME del mercado argentino, dedicadas al desarrollo de software.

Descripción de los estudios realizados:

Entrevistas con expertos:

Fue realizada una extensa entrevista al fundador y Presidente de una reconocida empresa argentina dedicada al desarrollo de software, a la consultoría, y otros servicios relacionados. Esta persona fue seleccionada para ser entrevistada por ser uno de los grandes referentes de la actividad en Argentina, con muchos años de experiencia nacional e internacional. Los conceptos principales extraídos de esta entrevista están sintetizados en los Anexos, conceptos que fueron además de gran utilidad y tenidos en cuenta durante el avance de esta investigación.

Encuestas a Colegas de la Industria.

Fue conducida una encuesta on-line a colegas de la industria nacional del software sobre la que se obtuvieron 50 respuestas de personas que desempeñan su actividad en diferentes empresas y posiciones. Los resultados de esta encuesta están resumidos en los Anexos y las respuestas fueron tenidas en cuenta durante el avance de esta investigación y la formulación de las conclusiones.

Articulación:

Este trabajo de investigación está dividido en 8 capítulos y 3 Anexos:

- Capítulo 1. Productividad General.
- Capítulo 2. Productividad en Desarrollo de Software.
- Capítulo 3. Factores que Afectan la Productividad.
- Capítulo 4. Relación entre Productividad y Calidad.
- Capítulo 5. La Productividad en Términos del Negocio.
- Capítulo 6. Metodologías y Productividad.
- Capítulo 7. Uniendo los Mundos en un Nuevo Enfoque.
- Capítulo 8. Conclusiones y Recomendaciones.
- Anexos 1, 2 y 3. Bibliografía, Resultados de la Encuesta y Entrevistas.

CAPITULO 1. PRODUCTIVIDAD GENERAL

Este trabajo está referido específicamente al estudio de la productividad en actividades de desarrollo de software y a la determinación de los factores que la afectan de manera positiva o negativa. Pero para comenzar a hablar de productividad en desarrollo de software, es necesario primero definir y explicar el concepto de la productividad general que se utiliza, por ejemplo, en la industria. Si se logra definir y entender de manera clara la productividad como un concepto general, será luego más fácil comprenderla en el marco de la actividad en la que se enfocará este trabajo.

Una primera definición dice que la productividad (P) es la relación entre la cantidad de producto o servicio generado, y los recursos utilizados para su producción. John G. Belcher[1] en su libro Productividad Total dice que: "*El concepto de productividad es bastante simple: se trata de la relación entre lo que produce una organización, y los recursos requeridos para tal producción*".

Productividad = Producto Obtenido / Recursos utilizados.

Muchas veces el concepto de productividad se confunde con el de producción, siendo que son cosas totalmente diferentes, donde la producción se refiere solamente a la cantidad de producto generado, y es uno de los términos que permiten calcular la productividad.

Para dejar claros los conceptos que se relacionan con la productividad se dice que **producto (o producción)** es lo que se obtiene o lo que se espera obtener, los **recursos** que se emplean para obtener estos resultados es lo que se llama **insumos**. La **eficacia** es la obtención, o la capacidad de obtener los resultados deseados, y la **eficiencia** se logra si se puede obtener el resultado esperado con el mínimo consumo posible de recursos.

[1] John G. Belcher, autor de varios libros de temas empresariales, en su libro Productividad Total dice que la Productividad Total sólo puede ser lograda cuando se concibe como un proceso de gestión, optimizando la totalidad de los recursos de la organización.

Con estos conceptos aclarados, es posible dar otras definiciones matemáticas de productividad:

- Productividad = Producto / Insumos.
- Productividad = Eficacia / Eficiencia.
- Productividad = Valor para el cliente / Costo para el productor.

Siendo que para la producción de un determinado producto o servicio suelen utilizarse varios recursos o insumos diferentes (insumos físicos, mano de obra, energía, espacio, tiempo, etc.), la productividad puede ser calculada teniendo en cuenta un solo recurso (productividad parcial o mono-factorial), o teniendo en cuenta todos los recursos que se utilizan (productividad total multi-factorial).

La primera relaciona la producción obtenida con uno solo de los recursos utilizados y se obtiene de ella un análisis parcial e incompleto que puede ser de utilidad para cierto tipo de análisis; la segunda relaciona la producción obtenida con la sumatoria de los diferentes recursos utilizados, expresados estos en una denominación común.

Muchos trabajos tratan por separado la productividad laboral, que considera el uso de la fuerza laboral (horas hombre) en la relación con el producto obtenido, y sobre la que se entrará en detalle más adelante en este capítulo.

La Productividad Parcial o Mono-factorial:

La productividad es una relación entre producción y recursos, donde en el caso de la productividad parcial se considera uno solo de los recursos utilizados. Es posible aplicar esta relación para el caso de una empresa, un proceso productivo, una línea de producción, una máquina, e inclusive una persona o un grupo de personas. En este caso, productividad (P) resulta de dividir la cantidad de producto obtenido (Pro), por la cantidad de un recurso utilizado (R):

P = Pro / R

Productividad (P) es entonces un coeficiente de producto por unidad de insumo o recurso utilizado. Por ejemplo, una máquina que utiliza 10 metros de aluminio para fabricar 1000 latas de gaseosa, tiene una productividad de:

P = 1000 latas / 10 metros = 100 latas por metro.

Para el mismo caso, una máquina más eficiente que reduce el desperdicio de material, podría producir 1200 latas con 10 metros de aluminio, obteniendo una productividad de:

P = 1200 latas / 10 metros = 120 latas por metro.

Claramente ambas máquinas producen el mismo producto final, latas de gaseosa, utilizando el mismo insumo, aluminio, pero una lo hace de manera más eficiente que la otra ya que es capaz de fabricar más latas con la misma cantidad de aluminio. Esta máquina hace un uso más eficiente[2] de los recursos, o lo que es lo mismo, logra una mayor productividad.

También es posible medir la productividad en función del tiempo utilizado para la realización de un producto, siendo que el tiempo puede ser considerado un recurso, en cuyo caso productividad es:

P = Producto / Tiempo

Llevado esto al ejemplo de las latas de gaseosa, si una máquina produce 30000 latas de gaseosa en 1 semana de producción continua se dice que esta máquina tiene una productividad de:

P = 30000 latas / (7d * 24h) = 30000 latas / 168 horas = 178,5 latas por hora.

[2] Enrique Hernández Laos, Profesor–investigador de la Universidad Autónoma Metropolitana de la República de Méjico, publicó en 2007 un estudio titulado La productividad multifactorial: concepto, medición y significado, donde menciona que: un proceso (o una máquina) es "eficiente" si, dado un conjunto de insumos, el proceso (la máquina) es capaz de generar la máxima producción posible, que suele estar determinada por la máxima capacidad alcanzable por unidad de tiempo.

Una máquina más veloz, que produce 32000 latas 1 semana de producción continua, tendría una productividad de:

P = 32000 latas / (7d * 24h) = 32000 latas / 168 horas = 190,4 latas por hora.

Mismo ejemplo podría ser utilizado para calcular la productividad respecto de cualquiera de los recursos utilizados para la producción de las latas de gaseosa: energía eléctrica, horas hombre, combustibles si aplicara, u otros insumos.

Productividad Total o Multifactorial (respecto varios insumos o recursos)

En la realidad, son muchos los factores que afectan la productividad, y muchas las variables que hay que considerar para su cálculo. La medición de la productividad parcial o mono-factorial, es útil para un análisis parcial, respecto de una sola variable, que solo es válido cuando se asume que las otras variables se mantienen constantes, cosa que en la práctica suele no ser verdad. Es por eso que el estudio de la productividad parcial podría conducir a graves errores de interpretación de la realidad y a tomar decisiones incorrectas.

Resulta entonces muy importante poder estudiar la productividad, teniendo en cuenta todas las variables que intervienen en su cálculo para poder realizar un análisis completo y adecuado.

Tomando a la industria como ejemplo típico, los factores que podrían intervenir para un cálculo completo de productividad son: Materia Prima utilizada, Energía que se consume, el costo de la mano de obra, el capital utilizado o su amortización, costos edilicios, alquileres, otros insumos. Teniendo en cuenta estos factores, el cálculo de la productividad sería:

P = Pro / R

P = Producto / (Materia Prima + Energía + Capital + Trabajo + Otros Insumos)

Siendo que las variables que componen el denominador se contabilizan en diferentes unidades incompatibles para una sumatoria, lo que se hace normalmente es expresarlas en términos monetarios y sumarlas, lo que implica conocer el costo de cada insumo o recurso.

Volviendo al ejemplo anterior de las latas de gaseosa, se podrían tomar los siguientes datos para componer la fórmula del cálculo de la productividad multi-factorial para un día de trabajo:

- Producción:
  - Cantidad de Latas Producidas: 4250.
  - Precio en el Mercado: $1000.-

- Insumos / Recursos:
  - Materia Prima (Aluminio): 40 metros = $100.
  - Energía Eléctrica: 1Kw = $25.
  - Costo de la Hora Hombre: $15
  - Horas Hombre: 24 (3 operarios en 3 turnos de 8 horas).

Con estos valores se puede realizar el cálculo de la productividad multifactorial de la siguiente manera:

P = 4250 latas / ($100 + $25 + ($15 * 24)) = 4250 latas / $485
P = 8,7 latas por cada peso de costo.

O también:

P = $1000 / ($100 + $25 + ($15 * 24)) = $1000 / $485
P = 2,06 pesos de valor por cada peso de costo.

Ambas formas indican el producto obtenido (cantidad o valor) para una determinada cantidad de recursos utilizados para su producción. Los recursos o insumos se expresan en términos monetarios para poder sumarlos.

La productividad medida en términos monetarios puede verse influida por las oscilaciones de los costos de los insumos, o por el precio de venta del producto. Por eso es importante que se mida la productividad siempre bajo hipótesis de precios constantes, o de variaciones regulares para todos.

Otro tema a tener en cuenta es que la productividad no da información sobre los recursos que permanecen ociosos en la empresa. Al medir la productividad de una línea de producción por ejemplo, se tienen en cuenta los recursos afectados a la misma. Si se aplica esta misma fórmula para todas las líneas de producción de una empresa se puede cometer el error de dejar de lado algunos recursos ociosos que no estén afectados a las líneas de producción. Los valores individuales de productividad de cada línea de producción podrían ser muy buenos, pero irreales desde una perspectiva de productividad global de la empresa.

La importancia de la Productividad

Por lo expuesto anteriormente se concluye que la productividad es un indicador económico directo y de gran importancia que dice cuán eficiente es una empresa, (o una línea de producción, o una máquina, o un grupo de personas, etc.), en el uso de los recursos y del tiempo para lograr el producto deseado. La productividad tiene relación directa con la rentabilidad, indicador fundamental y razón de ser de las empresas.

En un entorno altamente competitivo como son las economías actuales, la productividad puede marcar claramente la diferencia entre el éxito o el fracaso de una empresa o alguna de sus unidades de negocio. Es por eso que resulta muy importante para la gestión de las empresas el poder medir y conocer la productividad, como paso inicial para gestionarla y mejorarla. Muchas empresas no tienen consciencia de esto y no realizan mediciones de productividad, o las realizan de manera incompleta, lo que muchas veces puede llevarlos a conclusiones erróneas que seguramente resultarán en decisiones equivocadas.

Medir y controlar de forma permanente la productividad, junto con otros indicadores como son los costos, la calidad, el nivel de los servicios y la satisfacción de los clientes, es fundamental si se pretenden lograr ventajas competitivas sostenibles en un mundo de constantes cambios tecnológicos, económicos, políticos y sociales que afectan y desafían a todas las empresas.

La productividad del trabajo o laboral

Uno de los tipos de productividad de los que se suele hablar en los estudios sobre este tema es el de la productividad del trabajo, o productividad laboral. Se trata simplemente de aquella productividad en la que se busca determinar el nivel de eficiencia de la fuerza laboral en sus actividades de producción[3].

La productividad laboral puede medirse utilizando la cantidad o el costo de las horas de trabajo (horas hombre) como divisor en la fórmula, y dará como resultado la cantidad o valor de producto generado por hora hombre.

P = Producto / Horas Hombre
P = 500 televisores montados / (4 hombres * 1 semana)
P = 500 televisores montados / 160 horas hombre
P = 3,12 televisores montados por hora hombre.

Lo complejo del cálculo y del estudio de la productividad laboral, reside justamente en la naturaleza humana del recurso, y las variaciones que se presentan en la productividad por factores netamente humanos como la motivación, el estado de ánimo, el nivel de entrenamiento de la gente, el nivel de ausentismo, o la simple diferencia que se genera cuando diferentes personas realizan la misma tarea (velocidad, precisión, etc.).

[3] Esta definición es dada por Enrique Hernández Laos en el mismo trabajo citado antes, aunque menciona que "aumentos significativos del producto por hombre ocupado pueden estar reflejando no solamente una mejor utilización de los esfuerzos laborales del país, sino también pueden ser consecuencia de un proceso de sustitución factorial".

Así como podría resultar muy simple calcular qué mejora en la productividad es posible obtener mediante el cambio de una máquina en la línea de producción, quizá la simple lectura de las especificaciones técnicas responda la pregunta; en el caso de las personas, ¿qué tan fácil puede ser estimar la mejora en la productividad que generará una capacitación, o un aumento salarial que se supone que mejorará la motivación?

Algunas actividades, por ejemplo las de fabricación manual o artesanal, la construcción; o aquellas actividades basadas en la creatividad, el talento y el conocimiento, como la producción de servicios, la consultoría, o el desarrollo de software entre otras, tienen esta complejidad adicional de medir y gestionar la productividad de las personas, con lo complejo y desafiante que esto resulta.

El ciclo de la productividad

Medir y mejorar la productividad implica la realización por parte de cualquier empresa de una actividad cíclica que se puede dividir en 4 fases o etapas. Es lo que comúnmente se llama el ciclo de la productividad[4]. Este ciclo debe realizarse de manera ininterrumpida para asegurar su efectividad y utilidad, cómo cualquier otro ciclo de mejoras.

Las 4 etapas del ciclo de la productividad son:

- **Medir**: se trata de calcular el valor actual de la productividad, para lo cual es necesario contar con los resultados de las mediciones de uso de recursos e insumos en el proceso productivo que se está evaluando.
- **Evaluar**: se trata de analizar los resultados obtenidos, compararlos con los objetivos planteados, con valores de referencia o valores esperados. Se pueden analizar y comparar respecto de valores standard de mercado, valores de la competencia, o cualquier otro parámetro o criterio que indique si la productividad debe ser mejorada y en qué medida.

[4] El ciclo de la productividad dividido en 4 etapas es parte del Modelo de Productividad Total (TPM) que fue desarrollado por David Sumanth en el año 1979 y publicado en su libro titulado "Administración para la Productividad Total".

- **Planificar**: planificar tiene que ver con definir las acciones que se realizarán para mejorar la productividad, y establecer los objetivos a ser alcanzados. Cómo, cuándo, quién y a qué costo, son preguntas que se deben responder en esta etapa respecto de la forma en la que se busca mejorar la productividad de la empresa.
- **Mejorar**: se trata de ejecutar el plan trazado para la mejora de la productividad, que seguramente tendrá que ver con realizar cambios en los procesos productivos de la empresa para lograr los objetivos definidos.

Este ciclo de 4 etapas se realiza generalmente como un continuo, donde las 4 etapas se entremezclan y se intercalan para diferentes iniciativas y acciones de mejora que se dan en forma simultánea en una empresa. Lo importante es que ocurra sin interrupciones, y que se cumpla el objetivo de cada etapa, como parte de una inercia cultural de la organización en búsqueda de la mejora.

La clave para poner en práctica la gestión de la productividad radica en actuar y medir de manera simultánea el impacto de los cambios realizados sobre todos y cada uno de los componentes que participan en el proceso productivo.

Para gestionar y mejorar la productividad, es fundamental la capacidad que tengan las empresas de medir los recursos que utilizan y medir también los productos que generan, cosa que cambia de industria a industria y de actividad en actividad. En este aspecto, se ven puntualmente desafiadas aquellas empresas que utilizan o producen intangibles. El concepto de la **intangibilidad de las cosas** será tratado más adelante en este trabajo.

CAPITULO 2. PRODUCTIVIDAD EN SOFTWARE

Habiendo aclarado en el capítulo anterior el concepto de la productividad en general, este capítulo definirá el concepto de la productividad, aplicada a las actividades de desarrollo de software.

Como en cualquier otra actividad, para calcular la productividad en actividades de desarrollo de software, es necesario establecer la relación matemática (división) entre el producto generado, que en este caso es software, y los recursos utilizados para su construcción, que a pesar de ser varios, el recurso humano es el principal y preponderante en la ecuación de costos de cualquier actividad de desarrollo de software. Por tratarse de personal calificado que emplea su capacidad y conocimiento para la creación del producto, y en general con sueldos altos respecto de otras industrias, resulta que los otros costos que se pueden afectar a la actividad de desarrollo de software, son despreciables respecto del costo laboral. Por esto, este trabajo se enfocará en los aspectos humanos y metodológicos de la actividad, y dejará de lado otros costos como los edilicios, energía eléctrica, adquisición de equipos, etc.

Aquí comienzan a aparecer los aspectos complejos propios del mundo del software, y que en muchos casos se relacionan con el concepto de la intangibilidad de las cosas que fue mencionado en el capítulo anterior y la capacidad que cada empresa tenga para medirlos en el marco de su actividad. Se relacionan también con la naturaleza humana del principal recurso utilizado y la complejidad que aflora cuando se intentan entender los factores que afectan su productividad y cómo mejorarla.

Se analizan a continuación cada uno de los 2 factores que componen la fórmula de la productividad, y un tercer factor relacionado al alcance del estudio y cálculo, iniciando por el tercero:

- **Alcance o Scope del estudio**: la productividad puede ser calculada para una empresa entera, para una línea de producción, para una máquina o para un sector; en el mundo del desarrollo de software también se puede calcular y estudiar la productividad para una empresa, para una división, para un grupo particular que se dedica a desarrollar determinado producto o tecnología, para un determinado proyecto, etc. Este aspecto no presenta mayor complejidad, solo prestar atención y decidir cuál será el alcance del estudio, teniendo en cuenta que diferentes grupos, proyectos o productos tienen diferentes propiedades o particularidades que los hacen únicos e incomparables. Por ejemplo, no sería lógico intentar comparar la productividad de un grupo que desarrolla algoritmos de bajo nivel en lenguajes primitivos, con otro que desarrolla aplicaciones gráficas para la web, ya que muy posiblemente sus métodos, herramientas de trabajo, tecnologías, y las funcionalidades y valores de los productos resultantes serán totalmente dispares.
- **El tamaño o el valor del Producto**: aquí se plantea un dilema interesante, respecto de la cantidad y/o valor de producto construido, siendo que el software es un intangible que no permite medir de forma certera la cantidad, el tamaño, la funcionalidad, y/o el valor de lo construido por un equipo de desarrollo en un determinado punto del proceso. Volviendo al ejemplo de las latas de gaseosa, se puede saber en todo momento cuantas latas hay, es cuestión de simplemente contarlas a la salida de la máquina y listo. A cada semana, a cada día, a cada hora, es posible saberlo y tomar acciones para ajustar la productividad si el valor obtenido no es el deseado. Pero en el software no es igual. ¿Cómo es posible saber cuánto software han construido los desarrolladores al final de un día, una semana o un mes? ¿Cómo se puede saber a ciencia cierta el grado de avance para conocer si la productividad es la esperada y saber de antemano si el proyecto llegará a término? ¿Cómo es posible saber la cantidad de fallas que se generaron y la cantidad de retrabajo que habrá al final del proyecto para solucionarlas? Muchas preguntas que surgen a partir de la idea planteada antes de la intangibilidad de las cosas, que aplica perfectamente al software porque es un intangible.
- **La cantidad de recursos o insumos**: este aspecto, cómo el primero, tampoco presenta gran complejidad respecto de su aplicación en la fórmula de la

productividad, ya que se trata mayormente de trabajo realizado por el personal afectado a las actividades de desarrollo de software en sus diferentes especialidades o perfiles: programadores, analistas, diseñadores, gerentes de proyecto, arquitectos, especialistas en comunicaciones, etc., y que es posible medir en horas hombre dedicadas a cada proyecto con sus respectivos costos. Lo complejo de este punto es que hay muchos factores que pueden variar la productividad de la gente, los factores humanos como la motivación, el nivel de entrenamiento, el estado de ánimo, las relaciones interpersonales, y una suma de factores externos que pueden afectar la productividad de las personas, desde el ambiente de trabajo, hasta la situación económica y social del país y las cuestiones personales.

De estos tres factores descriptos, se desprenden al menos 3 grandes preguntas que este trabajo intentará responder, en este capítulo o en los próximos:

- ¿Cómo es posible medir la cantidad y/o el valor del producto construido, siendo que el software es un intangible?

- ¿Cómo se puede componer un índice de productividad que sirva de base para estudiarla y mejorarla, pudiendo medir el impacto de las acciones de mejora sobre los procesos, métodos, tecnología, y también sobre las personas?

- ¿Qué se puede hacer para que las horas hombre trabajadas en un proyecto de desarrollo de software rindan al máximo, logrando construir el producto esperado con la menor cantidad de horas posibles?

Para comenzar a buscar la respuesta a estas preguntas, esta investigación entrará en el mundo de las mediciones de software. Este es un tema sobre el que se ha investigado y escrito mucho, pero que no todas las empresas de software toman en serio y valoran respecto del beneficio que podría traerles al permitirles tomar decisiones basadas en datos concretos y no en sensaciones. Las empresas líderes del mundo en desarrollo de software,

también son líderes en su capacidad de colectar, estudiar y usar mediciones para la gestión de sus actividades y para la mejora de sus procesos y procedimientos[5].

Aún en 2013, donde el estado del arte de las cosas ha alcanzado una evolución impresionante; donde la ciencia y la tecnología son claros protagonistas del desarrollo mundial; donde la informática y el desarrollo de software han alcanzado resultados que hace unos años parecían de ciencia ficción; aún hoy los proyectos de desarrollo de software sufren algunos de los mismos problemas que hace 20 o 30 años: grandes atrasos, importantes desvíos de costos, bajos niveles de calidad e insatisfacción de los usuarios.

En el libro "Applied Software Measurement" de Caper Jones[6], figura esta frase que resulta muy ilustrativa de este concepto, escrita por el autor del prólogo del libro, Doug Brindley, Presidente y CEO de Software Productivity Research, LLC:

Measure to Know
Know to Change
Change to Lead

Es tan importante para las empresas que se dedican a desarrollo de software, o para empresas que tienen actividades de desarrollo de software que son representativas en sus negocios, tener buenos esquemas de mediciones, así como también es importante tener buenas herramientas y metodologías de desarrollo. Ambos, métodos y mediciones deben trabajar en conjunto. Sin un buen esquema de mediciones, el resultado de excelentes metodologías y herramientas podría quedar imperceptible, o podría no convencer al top management de la empresa al no poder mostrar los resultados en números concretos; de la misma forma, un muy buen esquema de mediciones solo mostraría constantes desvíos de

[5] Concepto mencionado por Caper Jones en Applied Software Measurement. Ver nota siguiente.

[6] Caper Jones: es un conocido investigador y especialista americano en ingeniería de software, autor de conocidas publicaciones como “Software Engineering Best Practices” y “Applied Software Measurement”. Dedicó gran parte de su carrera a colectar y analizar datos sobre calidad de software, mejores prácticas, riegos, estimaciones y software Project Management, que fueron usados por muchas organizaciones y educadores.

costos, atrasos y problemas de calidad del software sin métodos y herramientas adecuados para realizar la actividad. Ambos, metodologías y mediciones son importantes y deben ser seleccionados e implementados en conjunto, para poder lograr buenos resultados respaldados por información correcta.

¿Pero qué medir? Es una gran pregunta, que tiene una respuesta por demás simple. Para saber qué medir en actividades de desarrollo de software, como en cualquier otra actividad, basta con mirar qué miden las empresas líderes de la industria. Estas empresas que se caracterizan por lograr excelentes resultados, pueden enseñar mucho respecto de sus métodos, y qué información utilizan para conocer y mejorar su funcionamiento.

Un uso no menos importante que puede dársele al cálculo de productividad en el desarrollo de software, es el de poder establecer valores promedio o estándares para la industria. Muchas empresas son parte de una industria o parte de un segmento de la industria que está formado por empresas similares que realizan la misma actividad. Resulta interesante poder establecer valores que permitan saber si una empresa está trabajando dentro del rango normal de su industria, por debajo o por arriba. Esta información puesta a disposición de los clientes, sería de gran ayuda al posibilitar la evaluación de sus posibles proveedores en términos de la productividad que obtendrán.

Sobre la primera de las 3 preguntas planteadas más arriba:

- ¿Cómo es posible medir la cantidad y/o el valor del producto construido, siendo que el software es un intangible?

Las siguientes son diferentes formas encontradas en la literatura experta, respecto de cómo medir el tamaño, el valor funcional o el valor económico del producto de software construido. Cada forma fue analizada determinando sus ventajas y desventajas, y sus posibilidades de implementación:

LOC (Lines of Code):

El software se "escribe" en diferentes lenguajes de programación de computadoras, formados estos por instrucciones que se listan una debajo de la otra formando un programa. Cada programador escribe sus líneas de código que luego somete a un proceso de compilación para construir la pieza de software deseada: ejecutable, librería, etc. Cuanto más complejo sea el software a construir, más tiempo les llevará a los programadores y más líneas de código serán escritas. Claramente, la cantidad de líneas de código es una medida de la complejidad y del trabajo realizado.

Resulta ser obvio. ¡Problema resuelto! Se puede utilizar la cantidad de líneas de código escritas y la cantidad de horas-hombre trabajadas para calcular la productividad de un proyecto o un equipo, la cual será presentada en Líneas de Código por Hora Hombre:

$$P = LOC / HH.$$

Un equipo altamente productivo realizará más LOC por hora hombre que otro equipo no tan productivo. Con este cálculo de productividad se podrá estimar cuanto falta para terminar un proyecto, y hasta se podrá medir el impacto de acciones de mejora sobre los procesos y las herramientas. Parece perfecto, pero no lo es.

Las primeras mediciones de tamaño y productividad en desarrollo de software comenzaron allá por los años 50 (junto con la actividad), tomando la cantidad de líneas de código como medición de tamaño del software. Por aquel entonces el método era adecuado ya que solo se usaban pocos lenguajes de programación de bajo nivel para aplicaciones de índole militar o científica, los programas eran pequeños (del orden de las 1000 líneas de código) y en la mayoría de los casos eran construidos en un solo lenguaje. Los proyectos eran comparables de esta forma, observando simplemente algunas reglas para el conteo de líneas de código.

A partir de los años 60 se comenzó a complicar la medición con LOC ya que aparecieron lenguajes de más alto nivel como FORTRAN o COBOL que permitían desarrollar sistemas con mayor funcionalidad, y las computadoras comenzaron a utilizarse en el mundo de los negocios con sistemas que comenzaban a contarse en cientos de miles de líneas de código. Ya no era tan fácil el cálculo de la productividad con el simple conteo de las líneas de código[7].

Esta aparición de los lenguajes de más alto nivel puso de manifiesto una de las grandes debilidades de la medición por LOC, y es que esta tiende a privilegiar a los lenguajes de más bajo nivel. Por ejemplo, si se comparan 2 proyectos que produjeron 1000 líneas de código en 1 semana hombre, uno de ellos en lenguaje Assembler y el otro en COBOL; la productividad de ambos sería de 1000 LOC / 40 horas hombre, lo que es igual a 25 LOC/Hora. Pero cómo es sabido, 1000 líneas de código en COBOL dan mucha más funcionalidad que 1000 líneas de código en Assembler. Esto para LOC es invisible y el proyecto en Assembler se muestra igual al proyecto en COBOL.

Además comenzaron a realizarse otras actividades en el marco de los proyectos de desarrollo de software que no eran representadas por las LOC: se trata de las actividades de análisis funcional, diseño y documentación del software, que resultaban invisibles en la medición por LOC.

Hoy la realidad es aún más compleja. Hay cientos de lenguajes diferentes, con instrucciones y estilos de programación que los hacen incomparables mediante el simple conteo de líneas de código. Hay entornos y herramientas de desarrollo que realizan generación automática de código a partir del trabajo de un programador con una herramienta gráfica. Además es muy común que un mismo producto de software sea construido usando diferentes lenguajes para sus componentes.

[7] Capers Jones menciona en Applied Software Measurement: "*An obvious issue with lines of code metrics is that there are more than 700 programming languages and dialects in use circa 2008. Of the approximately 700 programming languages and dialects in use today, there are formal code counting rules for less than 100*".

Hoy, la medición del tamaño del software mediante LOCs presenta varias debilidades que hacen que su uso sea solo viable a proyectos y situaciones muy específicas:

- Tendencia de LOC a privilegiar a los lenguajes de bajo nivel.
- Lenguajes de diferentes niveles hacen que los valores de LOC no sean comparables (Ej.: 1000 líneas de C++ no pueden ser comparadas con 1000 líneas de PL/SQL).
- Muchos lenguajes diferentes utilizados en el mismo proyecto/producto.
- Generación automática de código hecha por herramientas de diseño.
- Tendencia cada vez mayor al reúso de código que LOC no considera.
- Actividades que no sean escribir código quedan invisibles a la medición.
- Estilos personales de programación pueden distorsionar la medición.

Todas estas razones hacen que la medición por LOC (Lines of Code) sea inaplicable para cálculos de productividad y queda descartado en este estudio.

Puntos de Función:

Ante los problemas y debilidades que ya presentaba la medición LOC, a mediados de los años 70, IBM desarrolló una nueva forma de medir el tamaño del software llamada puntos de función. Esta forma comenzó a hacerse popular en todo el mundo, y en los primeros años de la década del 80 se formó una organización sin fines de lucro llamada IFPUG[8] (International Function Point Users Group). Esta organización se encargó de continuar con el desarrollo y la popularización del método, llegando a tener organizaciones afiliadas en la mayoría de los países desarrollados del mundo.

El mismo IFPUG describe el método en su Web Site (http://www.ifpug.org) de esta manera:

[8] IFPUG – International Function Point Users Group: es una organización internacional, sin fines de lucro, basada en Estados Unidos que fue creada en 1986 con el fin de mantener, desarrollar y difundir el método llamado FPA (Function Point Analysis). Su web es http://www.ifpug.org/

Function Point Analysis (FPA) is a sizing measure of clear business significance. First made public by Allan Albrecht of IBM in 1979, the FPA technique quantifies the functions contained within software in terms that are meaningful to the software users. The measure relates directly to the business requirements that the software is intended to address. It can therefore be readily applied across a wide range of development environments and throughout the life of a development project, from early requirements definition to full operational use. Other business measures, such as the productivity of the development process and the cost per unit to support the software, can also be readily derived. The function point measure itself is derived in a number of stages. Using a standardized set of basic criteria, each of the business functions is a numeric index according to its type and complexity. These indices are totaled to give an initial measure of size which is then normalized by incorporating a number of factors relating to the software as a whole. The end result is a single number called the Function Point index which measures the size and complexity of the software product.

Como se puede apreciar en el texto transcripto, el método de Puntos de Función no es una medida de tamaño del software como lo es LOC, sino que es una medida de la funcionalidad que el software posee para el usuario final. Se puede hablar también de tamaño, pero en este caso se trata de tamaño funcional. Mide cuanta funcionalidad un producto de software le brinda a sus usuarios en términos de funcionalidad aplicada al negocio. De hecho, el cálculo de los puntos de función de un producto de software se realiza a partir de los requerimientos funcionales de los usuarios o de las funciones del sistema, lo que supone que deben llevarse a cabo prácticas de ingeniería de requerimientos como base para el cálculo de los puntos de función.

En esencia, el método de puntos de función es un total ponderado de la funcionalidad y complejidad de un sistema, que se logra a partir de la clasificación de sus funciones en 5 tipos básicos y considerando su complejidad. Una de las principales ventajas y diferencias con LOC es que es independiente de la tecnología utilizada para el desarrollo, por lo cual es aplicable a proyectos que se desarrollarán en cualquier lenguaje de programación, e incluso

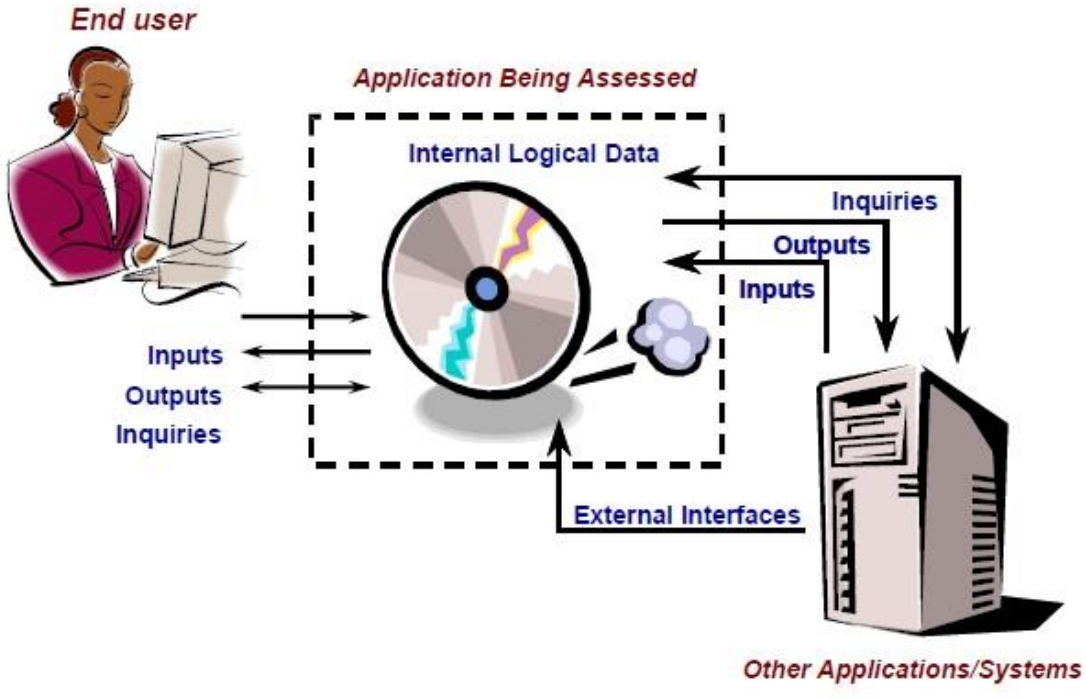
aquellos en los que se utilizarán varios lenguajes en forma simultánea. Permite también comparar productos y proyectos, aunque se utilicen distintas tecnologías para el desarrollo.

¿Cómo funciona?

Los siguientes son los pasos básicos para el conteo de los puntos de función[9]. Este trabajo no pretende realizar un análisis y explicación completa de este método, sino una descripción abreviada para poder entenderlo y analizar sus ventajas y desventajas:

1. El cálculo parte de los requerimientos funcionales del usuario, o la descripción de las funciones del sistema que deben ser primeros identificados y documentados. O sea que el primer paso para este análisis es construir o conseguir la lista de funciones a partir de entrevistas con usuarios, expertos del sistema o la lectura de su documentación.

Figura 1 - Análisis de Puntos de Función



Fuente: IFPUG, ¿*What are Function Points?*, www.ifpug.org, *p.* 1.

[9] La versión más reciente del método es la descripta en el Function Point Counting Practices Manual, versión 4.3.1 liberada por el IFPUG en Octubre 2009.

2. Cada requerimiento o función es clasificado en una de las siguientes cinco categorías:
    - Inputs (entradas): pantallas de ABMs, formularios, ventanas.
    - Inquiries (consultas): salidas de datos simples, respuesta inmediata.
    - Outputs (salidas): pantallas de resultados complejos, reportes, gráficos.
    - Internal Files (archivos propios): almacenamientos permanentes, tablas.
    - External Interfaces (interfaces con otros sistemas): exportación o importación de archivos, transacciones con otros sistemas, web services.
3. A cada requerimiento se le asigna un rango de complejidad que puede ser **low**, **medium** o **high** y se completa una tabla donde se totaliza la cantidad de funciones de cada tipo y complejidad.
4. Los valores obtenidos son multiplicados por un factor que el método determina para cada tipo de transacción y complejidad. La sumatoria total de lo obtenido es el valor de puntos de función no ajustados (UFP - Unadjusted Function Points).
5. Luego se calcula un factor de ajuste, que se logra ponderando 14 aspectos del sistema con un valor de 0 a 5, donde 0 indica que el factor no está presente y 5 indica que el factor es esencial para el sistema. Los aspectos a ponderar son:
    - Comunicación de datos
    - Procesamiento distribuido
    - Objetivos de rendimiento (performance)
    - Configuración del computador altamente utilizada
    - Respaldo y recuperación confiables
    - Ingreso de datos "on-line"
    - Actualización interactiva "on-line"
    - Interfaz compleja
    - Procesamiento complejo
    - Requerimientos de "re usabilidad"
    - Facilidad de conversión e instalación
    - Facilidad de operación
    - Cantidad de ubicaciones
    - Facilidad de realizar modificaciones

6. La sumatoria de los 14 factores ponderados se multiplica por 0,001 y se suma a 0,65 para obtener el factor de ajuste final. Esta forma asegura que se obtendrá un factor de ajuste que podrá variar entre 0,65 si todos los aspectos son 0 y 1,35 si todos son 14, ya que el método asume que el ajuste podrá ser como máximo de un 35% hacia abajo (sistema más simple) o hacia arriba (sistema más complejo).
    - FA = 0,65 + 0,01 * (fc1 + fc2 + fc3 ….. + fc14)

7. Finalmente se obtienen los puntos de función finales (o ajustados) multiplicando los UFP obtenidos en el paso 4, por el factor de ajuste obtenido en el paso 6.

A continuación se muestra un ejemplo de cálculo de Puntos de Función para un sistema cualquiera:

Las cantidades de funciones del sistema clasificadas por tipo y complejidad se vuelcan en la tabla 1, de donde se obtiene como sumatoria la cantidad de puntos de función sin ajustar (pasos 1, 2, 3 y 4):

Cuadro 1: Puntos de Función - Funciones por tipo y complejidad

	Low		**Medium**		**High**		
Elementos del Sistema	**Cantidad**	**Factor**	**Cantidad**	**Factor**	**Cantidad**	**Factor**	**Totales**
Inputs	5	3	3	4	4	6	**51**
Outputs	7	4	5	5	3	7	**74**
Inquiries	9	3	4	4	3	6	**61**
Internal Files	12	5	8	7	3	10	**146**
External Interfaces	0	7	3	10	1	15	**45**
						UFP	**377**

Los factores que están dados por el método.

Fuente: Ejemplo de elaboración propia.

Se calcula el factor de ajuste ponderando de 0 a 5 los 14 aspectos del sistema que el método considera para tal fin (paso 5):

Cuadro 2: Puntos de Función - Cálculo del factor de ajuste.

Aspecto del Sistema	Pond. (0 a 5)
Comunicación de datos	5
Procesamiento distribuido	5
Objetivos de rendimiento (performance)	3
Configuración del computador altamente utilizada	1
Respaldo y recuperación confiables	3
Ingreso de datos "on-line"	2
Actualización interactiva "on-line"	3
Interfaz compleja	1
Procesamiento complejo	3
Requerimientos de "re usabilidad"	2
Facilidad de conversión e instalación	1
Facilidad de operación	5
Cantidad de ubicaciones	1
Facilidad de realizar modificaciones	5
	40

Fuente: Ejemplo de elaboración propia.

Se obtiene el factor de ajuste y los puntos de función ajustados, que son el resultado del análisis (pasos 6 y 7):

FA = 0,65 + 0,01 * (fc1 + fc2 + fc3 ….. + fc14) = 0,65 + 0,01 * 34 = 1,05

AFP = 377 * 1,05 = **396**

Se obtienen entonces 377 puntos de función sin ajustar, que luego de aplicar el factor de ajuste quedan en 396 puntos de función ajustados.

El método de Puntos de Función o PFA (Funtion Point Analysis) está perfectamente detallado y explicado, junto con los criterios necesarios para el conteo de funcionalidades, identificación de los tipos de funcionalidades, los criterios para establecer los valores de complejidad de las funciones y la ponderación de los aspectos, en el documento llamado Function Point Counting Practices Manual, que el FPIUG desarrolla y mantiene y que hoy

está vigente en su versión 4.3.1. Este documento da las bases para un conteo correcto y uniforme y sobre el que deben capacitarse las personas u organizaciones para obtener buenos resultados.

Ventajas:

- Es consistente y comparable entre proyectos permitiendo generar diferentes métricas: costo por punto de función (productividad), errores por punto de función (calidad).
- Los puntos de función se pueden estimar desde el inicio del proyecto con los requerimientos, permitiendo estimar tiempos y costos.

Desventajas:

- Se necesita tener personal capacitado para lograr un conteo y clasificación correcta de las funcionalidades que resulte en una medición coherente.
- Como está basado en el conteo de funciones visible por el usuario, el método tiende a ocultar la complejidad de funciones internas del sistema que el usuario no ve, cómo los algoritmos de cálculo, que muchas veces son complejos y requieren de tiempo y esfuerzo en su implementación.

A pesar de sus desventajas, el método de Puntos de Función resulta adecuado para ser utilizado como parte de un estudio de productividad en desarrollo de software.

Use Case Points

Con la evolución de los métodos y lenguajes de programación, y especialmente con la adopción de la programación orientada a objetos, apareció un nuevo método de estimación de software llamado Use Case Points. Este método toma como base de conteo los Casos de Uso (Use Cases) que son parte del set de técnicas de modelado de UML y se usan para describir la funcionalidad de un sistema de software.

UML (Unified Modeling Language) es un lenguaje de modelado de software que incluye una serie de notaciones gráficas y varias técnicas para describir la funcionalidad y diseño de un software. Fue desarrollado por la empresa Rational Software (hoy IBM) en los años 90, y luego en 1997 fue adoptado y es administrado desde entonces por el Object Management Group (OMG)[10]. En el año 2000 fue aceptado como un standard de la industria por la ISO, constituyendo la norma ISO/IEC 19501:2005. La última versión de UML es la 2.4.1 publicada en el año 2011. Entre las notaciones y técnicas que UML incluye para describir un sistema se encuentran los casos de uso y los actores.

En el lenguaje UML, la funcionalidad del sistema se describe mediante los llamados Casos de Uso o Use Cases, que son listas de pasos que detallan una interacción entre el sistema descripto y un agente externo para cumplir una función u objetivo determinado. El agente externo en UML es llamado "actor", y puede ser una persona u otro sistema. La lista completa de los casos de uso de un sistema define su funcionalidad.

El método Use Case Point fue desarrollado en 1993 por Gustav Karner, un empleado de una empresa llamada Objectory que luego se convirtió en Rational Software, la que desarrolló UML. Cómo se verá más adelante, este método tiene una fuerte influencia del método de puntos de función, ya que se trata de una suma ponderada y luego ajustada por factores de complejidad que en este caso son técnicos y de ambiente.

¿Cómo funciona?

Use Case Points es la suma ponderada de la funcionalidad de un sistema, determinada por los casos de uso clasificados por complejidad, los actores del sistema también clasificados por complejidad, y luego todo ajustado por factores técnicos y ambientales. Este método que indica el tamaño funcional de un sistema, puede ser extendido a un método de

[10] El Object Management Group (OMG) es un consorcio formado en 1989 dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos, tales como UML, XMI, CORBA y BPMN. (http://www.omg.org).

estimación de costo de desarrollo en horas hombre si se lo completa con un factor de productividad.

Los componentes de la fórmula son los siguientes[11]:

- **UCP = Use Case Points**, el resultado final buscado, que indica tamaño funcional del sistema que se está estimando.
- **UUCP = Unadjusted Use Case Points**, que está formado por la suma de los UUCW (Unadjusted Use Case Weight) y los UAW (Unadjusted Actor Weight). No es otra cosa que el conteo de los casos de uso y los actores, antes ambos ponderados por un factor de complejidad que el método determina.
- **TCF = Technical Complexity Factors**, se trata de un factor de ajuste que se calcula ponderando 13 factores de complejidad técnica dados por el método, con pesos predefinidos, a los que se le asigna un impacto percibido por el equipo de proyecto, que puede varias entre 0 y 5, donde 0 indica que el factor no es relevante para el proyecto, 3 es promedio y 5 indica que el factor es de alto impacto. Se utilizan 2 constantes en la fórmula (C1 = 0,6 y C2 = 0,01) para lograr un valor que podrá variar entre 0,6 y 1,3. Esto hace que el TCF pueda ajustar el UUCP entre los límites de un -40% y un +30%.
- **ECF = Environmental Complexity Factor**, se trata de un factor de ajuste respecto del impacto que la experiencia del equipo de proyecto tendrá en la productividad. Se calcula ponderando 8 factores dados por el método, con pesos predefinidos, a los que se le asigna un impacto percibido por el equipo de proyecto, que puede tomar valores de 0, 1, 3 o 5; donde 1 es impacto muy negativo sobre la productividad, 3 es promedio y 5 implica un impacto muy positivo en la productividad. Se puede utilizar también el 0 (cero) para indicar que el factor no tiene impacto. También se utilizan 2 constantes en la fórmula (C1 = 1,4 y C2 = -0.03) para lograr un valor que podrá variar entre 0,425 y 1,4, lo que hace que el ECF pueda ajustar el UUCP entre

[11] Según lo describe Roy K. Clemmons, en su paper titulado *Project Estimation With Use Case Points*, y Ed Carroll en *Estimating Software Via Use Cases*.

un -57,5% y un +40%. Es importante destacar que el ECF tiene un impacto potencial muy alto sobre los UUCP, mayor que el impacto que podría tener el TCF.

La fórmula es: **UCP = UUCP * TCF * ECF**

Donde:

UUCP = UUCW + UAW

TCF = 0,6 + (0,01 * Technical Total Factor) **

ECF = 1.4 + (-0.03 * Environmental Total Factor) ***

**** Technical Total Factor** es la sumatoria de los 13 valores que resultan de multiplicar peso por complejidad de cada factor técnico.

***** Environmental Total Factor** es la sumatoria de los 8 valores que resultan de multiplicar peso por complejidad de cada factor ambiental.

El resultado UCP indica la cantidad de funcionalidad del sistema o proyecto, lo que permite en primera instancia tener una medida de comparación con otros proyectos que hayan sido estimados con el mismo mecanismo. Cómo se ha mencionado antes, el método puede ser ampliado para estimar las horas-hombre necesarias si el equipo de desarrollo cuenta con datos de productividad:

- **PF = Productivity Factor**, es un ratio de productividad que se obtiene de información histórica de la empresa o de la industria, e indica la cantidad necesaria típica (promedio) de horas hombre necesarias para la construcción de un UCP. Se utiliza para lograr una estimación de horas-hombre necesarias para el proyecto.

Horas Hombre Estimadas = UCP * PF

El método de estimación de tamaño funcional y esfuerzo por Use Case Points viene acompañado por valores predeterminados que permiten clasificar y dar peso ponderado a

los casos de uso, actores, factores de complejidad técnica y factores ambientales. Las siguientes tablas muestras estos valores dados que deben utilizarse en las estimaciones:

Cuadro 3: Use Case Points - Clasificación de Casos de Uso

Categoría	Descripción	Peso
Simple	Interface de usuario simple, que modifica solo 1 entidad de la base de datos. Es escenario exitoso tiene 3 pasos o menos y su implementación involucra menos de 5 clases.	5
Promedio	Diseño más complejo de la interface y modifica 2 o más entidades de la base de datos. Tiene entre 4 y 7 pasos y su implementación involucra entre 5 y 10 clases.	10
Complejo	Interface de usuario complejo o algoritmo de procesamiento. Modifica más de 3 entidades de la base de datos. Posee más de 7 pasos y su implementación involucra más de 10 clases.	15

Fuente: Roy K. Clammons, *Project Estimation with Use Case Points*, USA, 2006, *p.* 18.

Cuadro 4: Use Case Points - Clasificación de los Actores

Categoría	Descripción	Peso
Simple	El actor es otro sistema con una interface definida (tipo API).	1
Promedio	El actor es otro sistema con una interface abierta, tipo protocolo (Ej.: TCP/IP, HTMP o XML).	2
Complejo	El actor es una persona que interactúa con el sistema mediante una interface gráfica.	3

Fuente: Roy K. Clammons, *Project Estimation with Use Case Points*, USA, 2006, *p.* 19.

Cuadro 5: Use Case Points - Factores de Complejidad Técnica

Factor	Descripción	Peso
T1	Sistemas Distribuidos	2
T2	Performance	1
T3	Eficiencia del Usuario Final	1
T4	Procesamiento Interno Complejo	1
T5	Reusabilidad	1
T6	Fácil de Instalar	0,5
T7	Fácil de Usar	0,5
T8	Portabilidad	2
T9	Fácil de Mantener / Cambiar	1
T10	Concurrencia	1
T11	Características de Seguridad Especiales	1
T12	Provee acceso a terceros (clientes / Proveedores)	1
T13	Se necesitan instalaciones especiales para entrenamiento de usuarios	1

Fuente: Roy K. Clammons, *Project Estimation with Use Case Points*, USA, 2006, *p.* 20.

Cuadro 6: Use Case Points - factores de Complejidad Ambiental

Factor	Descripción	Peso
E1	Familiarizado con UML	1,5
E2	Trabajadores Part-Time	-1
E3	Capacidad de Análisis	0,5
E4	Experiencia en el Sistema	0,5
E5	Experiencia en Orientación a Objetos	1
E6	Motivación	1
E7	Lenguaje de Programación Difícil	-1
E8	Requerimientos Estables	2

Fuente: Roy K. Clammons, *Project Estimation with Use Case Points*, USA, 2006, *p.* 21.

A continuación se muestra un ejemplo paso a paso del cálculo de Use Case Points para un sistema cualquiera:

1. Se realiza el conteo de los casos de uso clasificados por complejidad, multiplicando luego cada cantidad por el peso dado. Se suman los resultados obtenidos para obtener los UUCW Totales.

Cuadro 7: Use Case Points - Conteo de Casos de Uso

Categoría	Descripción	Peso	Cantidad de Casos	Resultados
Simple	Interface de usuario simple, que modifica solo 1 entidad de la base de datos. Es escenario exitoso tiene 3 pasos o menos y su implementación involucra menos de 5 clases.	5	17	85
Promedio	Diseño más complejo de la interface y modifica 2 o más entidades de la base de datos. Tiene entre 4 y 7 pasos y su implementación involucra entre 5 y 10 clases.	10	12	120
Complejo	Interface de usuario compleja o algoritmo de procesamiento. Modifica más de 3 entidades de la base de datos. Posee más de 7 pasos y su implementación involucra más de 10 clases.	15	7	105
			UUCW Totales >>	**310**

Fuente: Ejemplo de elaboración propia.

2. Se realiza el conteo de los actores clasificados por complejidad, multiplicando luego cada cantidad por el peso dado. Se suman los resultados obtenidos para obtener los UAW Totales.

Cuadro 8: Use Case Points - Conteo de los Actores

Categoría	Descripción	Peso	Cantidad de Actores	Resultados
Simple	El actor es otro sistema con una interface definida (tipo API).	1	4	4
Promedio	El actor es otro sistema con una interface abierta, tipo protocolo (Ej.: TCP/IP, HTMP o XML).	2	2	4
Complejo	El actor es una persona que interactúa con el sistema mediante una interface gráfica.	3	8	24
			UAW Totales >>	**32**

Fuente: Ejemplo de elaboración propia.

3. Se documenta la complejidad percibida por el equipo de desarrollo para los 13 factores de complejidad dados, usando valores entre 0 y 5. Se multiplica luego el peso de cada factor por la complejidad percibida y se suman los resultados obtenidos para obtener el Factor Técnico Total.

Cuadro 9: Use Case Points - Ponderación de los Factores de Complejidad

Factor	Descripción	Peso	Complejidad (entre 0 y 5)	Factor Calculado
T1	Sistemas Distribuidos	2	1	2
T2	Performance	1	3	3
T3	Eficiencia del Usuario Final	1	3	3
T4	Procesamiento Interno Complejo	1	3	3
T5	Reusabilidad	1	0	0
T6	Fácil de Instalar	0,5	0	0
T7	Fácil de Usar	0,5	5	2,5
T8	Portabilidad	2	0	0
T9	Fácil de Mantener / Cambiar	1	3	3
T10	Concurrencia	1	0	0
T11	Características de Seguridad Especiales	1	0	0
T12	Provee acceso a terceros (clientes / Proveedores)	1	3	3
T13	Se necesitan instalaciones especiales para entrenamiento de usuarios	1	0	0
			Factor Técnico Total >>	**19,5**

Fuente: Ejemplo de elaboración propia.

4. Se determina el impacto que cada factor ambiental tendrá en el proyecto con valores 0, 1, 3 ó 5. Se multiplica luego el peso de cada factor por el impacto determinado y se suman los resultados obtenidos para obtener el Factor Ambiental Total.

Cuadro 10: Use Case Points - Ponderación de los Factores Ambientales

Factor	Descripción	Peso	Impacto (0, 1, 3 ó 5)	Factor Calculado
E1	Familiarizado con UML	1,5	5	7,5
E2	Trabajadores Part-Time	-1	0	0
E3	Capacidad de Análisis	0,5	5	2,5
E4	Experiencia en el Sistema	0,5	0	0
E5	Experiencia en Orientación a Objetos	1	5	5
E6	Motivación	1	5	5
E7	Lenguaje de Programación Difícil	-1	0	0
E8	Requerimientos Estables	2	3	6
			Factor Ambiental Total >>	**26**

Fuente: Ejemplo de elaboración propia.

5. Con los datos obtenidos en los pasos anteriores, se utilizan las fórmulas dadas por el método para calcular el valor de UCPs. Esto es:

UUCP = UUCW + UAW = 310 + 32 = 342

TCF = 0,6 + (0,01 * Technical Total Factor) = 0,6 + (0,01 * 19,5) = 0,795

ECF = 1,4 + (-0,03 * Environmental Total Factor) = 1,4 + (-0,03 * 26) = 0,62

UCP = UUCP * TCF * ECF = 342 * 0,795 * 0,62 = 168,5

6. Si se cuenta con un valor de índice de productividad (PF) ya sea un histórico de la empresa o un valor standard de mercado, se pueden estimar la cantidad de horas hombre que el proyecto demandará para su realización. Asumiendo un PF de 25 horas hombre por UCP, el resultado sería:

Horas Estimadas = UCP * PF = 168,5 * 25 = 4212,5 horas hombre

Es de destacar que en este caso, el valor inicial de 342 UCPs sin ajustar (UUCP) fue reducido por la valoración de los factores técnicos y ambientales que generaron en conjunto una reducción del -51% aproximadamente. En otro caso, si se indican factores de complejidad de alto impacto y características del equipo de proyecto negativas para la productividad, el valor inicial de UUCPs podría incrementarse en forma significativa. Es por este gran potencial de ajuste que resulta de suma importancia realizar una valoración correcta de los factores TCF y ECF.

Ventajas:

- Es un método que se basa en el conteo de funcionalidad, por lo cual puede ser utilizado para proyectos realizados en cualquier lenguaje de desarrollo, incluso aquellos en los que se utilizan varios en forma simultánea.
- Es consistente entre proyectos por lo que puede ser utilizado para comparar proyectos y realizar mediciones históricas de productividad.
- Puede ser usado para generar estimaciones tempranas, cuando se están desarrollando los casos de uso del sistema, mucho antes de comenzar a construir.
- Tiene en consideración a los “actores”, las personas o sistemas que interactúan con el sistema en cuestión.
- Tiene en consideración la experiencia del equipo de proyecto, que es un factor de ajuste muy importante que otros métodos, como puntos de función, no consideran.

Desventajas:

- Está asociado al modelo UML y a la generación de Casos de Uso, por lo que solo puede ser usado en proyectos que sigan estas técnicas.

- Para generar estimaciones precisas de costo (horas hombre), el método depende de tener datos históricos colectados que permitan calcular un factor de productividad (PF).

El método UCP (Use Case Points) es un método adecuado para realizar estimaciones y cálculos de productividad en aquellas empresas que utilizan UML y en especial Casos de Uso para describir la funcionalidad del software.

Método Delphi o Juico de Experto

El método Delphi, cuyo nombre se inspira en el antiguo oráculo de Delphos, fue ideado a comienzos de los años 50 en el Centro de Investigación de RAND Corporation[12] por Olaf Helmer y Theodore Gordon, como un instrumento para realizar predicciones sobre un caso de catástrofe nuclear. Desde entonces, ha sido utilizado frecuentemente como sistema para obtener predicciones y estimaciones sobre hechos futuros.

La técnica Delphi es un método de estructuración de un proceso de comunicación grupal que es efectivo a la hora de permitir a un grupo de individuos, como un todo, tratar un problema complejo.

Una sesión Delphi consiste en la selección de un grupo de expertos a los que se les pregunta su opinión sobre cuestiones referidas a acontecimientos del futuro. Las opiniones o estimaciones de los expertos se realizan en sucesivas rondas, anónimas, con el objeto de tratar de conseguir consenso, pero con la máxima autonomía posible por parte de los participantes. Es por eso que la capacidad de predicción del método Delphi se basa en la utilización sistemática de un juicio intuitivo emitido por un grupo de expertos. La selección del grupo de expertos que participará y la cuidadosa elaboración del cuestionario sobre el tema en cuestión, son actividades clave para lograr un buen resultado.

[12] RAND Corporation es una organización sin fines de lucro de investigaciones dedicada a desarrollar soluciones a problemas y desafíos públicos y globales en temas como economía, política, seguridad, terrorismo. El objetivo general es el de ayudar a construir comunidades y un mundo más seguro y próspero. Su sitio web es: http://www.rand.org/

El método está compuesto por 4 pasos generales:

- La formulación del problema.
- La selección de los expertos.
- La elaboración y el envío de los cuestionarios.
- Análisis de resultados y búsqueda de consenso.

Una de las principales desventajas del método es que en algún punto podría volverse algo largo, costoso y fastidioso en aquellos casos donde la búsqueda del consenso lleve a tener que emitir varios cuestionarios sucesivos. Afortunadamente, las nuevas tecnologías de comunicación y trabajo colaborativo han ayudado a mitigar este problema y han permitido un relanzamiento del método que había caído en desuso.

Se comenzó a utilizar en el campo del software a partir de 1981 por parte de Barry Boehm[13], quien a su vez le introdujo una mejora llamada Wideband Delphi o Delphi de Banda Ancha. A partir del uso dado por Boehm, el método Delphi ha sido utilizado ampliamente como forma de estimación en proyectos de desarrollo de software, tomando como base la opinión de los expertos acerca de la complejidad, tiempo y esfuerzo necesario para la implementación de las funcionalidades sobre las que se los consulta.

Existen 2 variantes del método utilizadas en software que se denominan Delphi Puro, el método original llevado a estimaciones de software; y Delphi de Banda Ancha o Wideband Delphi, que es la mejora introducida por Barry Boehm. La diferencia entre ambos reside en la posibilidad o no de que los expertos participantes intercambien opiniones entre sí, en medio de sus trabajos de análisis.

[13] Barry W. Boehm, es un ingeniero estadounidense nacido en 1935 que realizó importantes aportes al campo de la ingeniería de software, entre ellos el métodos de estimaciones COCOMO, el método espiral de desarrollo de software y varias publicaciones. Luego de su paso por varias compañías de software, y por el Departamento de Defensa de Estados Unidos, desde 1992 es profesor de ingeniería informática en el departamento de ciencias de la informática en la Universidad del Sur de California.

Delphi Puro:

- Un coordinador proporcionada a cada experto una especificación del proyecto propuesto y un formulario impreso o digital para volcar sus estimaciones.
- Los expertos completan el formulario de manera anónima. Pueden hacer preguntas al coordinador pero no entre ellos.
- El coordinador recibe las estimaciones, calcula los valores medios y envía a cada experto el valor medio de las estimaciones recibidas para una segunda vuelta de estimación.
- Los expertos deben realizar una nueva estimación anónima indicando las razones de las posibles modificaciones y/o sus divergencias respecto de los valores medios.
- Se repite el proceso hasta llegar a un consenso razonable.
- No se realizan reuniones en grupo en ningún momento del proceso.

Delphi de Banda Ancha (mejora de Barry Boehm):

- Un coordinador proporcionada a cada experto una especificación del proyecto propuesto y un formulario impreso o digital para volcar sus estimaciones.
- El coordinador reúne a los expertos para que intercambien sus puntos de vista sobre el proyecto.
- Luego de la reunión, los expertos completan el formulario de manera anónima.
- El coordinador recibe las estimaciones, calcula los valores medios y envía a cada experto el valor medio de las estimaciones recibidas para una segunda vuelta de estimación.
- Los expertos deben realizar una nueva estimación anónima indicando las razones de las posibles modificaciones y/o sus divergencias respecto de los valores medios.
- El coordinador convoca una reunión para que los expertos discutan las razones de las diferencias entre sus estimaciones.
- Luego de la reunión, los expertos completan nuevamente el formulario de manera anónima y lo envían al coordinador.
- Se repite el proceso hasta llegar a un consenso razonable.

La principal ventaja del método Delphi es que recoge la opinión de los expertos en el tema a tratar; mejor será el resultado obtenido cuanto mejor seleccionado esté el equipo que realizará la estimación. Es importante contar con un equipo lo suficientemente amplio y diverso para asegurar que muchas experiencias y puntos de vista diferentes serán considerados.

Respecto de un análisis de productividad, verán que el método Delphi no provee la información que se necesita, ya que no permite realizar por sí mismo una estimación del tamaño funcional del producto a construir. Se estiman tiempos y esfuerzos, pero no unidades de tamaño.

La combinación de Delphi con otros métodos de estimación de tamaño funcional puede dar la información necesaria para estudios de productividad.

Story Points y Planning Poker

Como en casi todos los aspectos de la vida en los cuales el pensamiento y el intelecto humano intervienen, en las actividades de desarrollo de software también comenzaron a diferenciarse varios estilos o corrientes de pensamiento y opinión, respecto de cómo deben realizarse estas actividades, y las ventajas y desventajas de diferentes formas.

En oposición al método de desarrollo de software tradicional, conocido como “waterfall”, que se caracteriza por procesos bien definidos, actividades reguladas, roles y responsabilidades asignados, etapas bien diferenciadas e importantes volúmenes de documentación que deben ser generados; comenzó a gestarse en los años 90 una corriente de pensamiento que proclamaba las ventajas de un estilo de desarrollo diferente, que ellos mismos llamaban “ágil”, destacando sus principales ventajas en el desarrollo del software en incrementos cortos, rápida entrega de valor al cliente mediante software funcional, equipos pequeños y auto-organizados con mucha interacción y comunicación, menor foco

en los procesos y en la documentación, involucramiento del cliente en el desarrollo y mucha flexibilidad para aceptar cambios en los requerimientos.

De la mano de esta forma de ver el desarrollo de software, nacieron los que hoy se denominan métodos ágiles, cuyos valores y principios fueron plasmados por 17 desarrolladores de software, grandes impulsores de esta corriente, que se reunieron en 2001[14] y publicaron el **Manifiesto para Desarrollo de Software Ágil**. Este documento destaca los 4 valores y los 12 principios que deben seguir los métodos de este estilo:

Los 4 valores del Manifiesto Ágil[15]:

- *Valorar más a los individuos y su interacción que a los procesos y las herramientas.*
- *Valorar más el software que funciona que la documentación exhaustiva.*
- *Valorar más la colaboración con el cliente que la negociación contractual.*
- *Valorar más la respuesta al cambio que el seguimiento de un plan.*

Los 12 principios del Manifiesto Ágil:

- *Nuestra principal prioridad es satisfacer al cliente a través de la entrega temprana y continua de software de valor.*
- *Son bienvenidos los requisitos cambiantes, incluso si llegan tarde al desarrollo. Los procesos ágiles se doblegan al cambio como ventaja competitiva para el cliente.*
- *Entregar con frecuencia software que funcione, en periodos de un par de semanas hasta un par de meses, con preferencia en los periodos breves.*
- *Las personas del negocio y los desarrolladores deben trabajar juntos de forma cotidiana a través del proyecto.*

[14] Del 11 al 13 de Febrero del 2001 se reunieron en el Snowbird Ski Resort, en las Montañas Wasatch Utah, un grupo de 17 desarrolladores de software experimentados con la idea de encontrar formas de desarrollar software alternativas a las tradicionales que ellos llamaban "heavyweight". Aquel grupo se autodenominó The Agile Alliance, y lo que surgió de la reunión es el Agile Software Manifesto, un documento público con los 12 principios de lo que a partir de ese momento se llamaría Agile Software Development. Aquel grupo de 17 desarrolladores estuvo formado por: Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland y Dave Thomas.

[15] El manifiesto del desarrollo de software Ágil, con sus 4 valores y 12 principios está publicado en http://www.agilemanifesto.org

- *Construcción de proyectos en torno a individuos motivados, dándoles la oportunidad y el respaldo que necesitan y procurándoles confianza para que realicen la tarea.*
- *La forma más eficiente y efectiva de comunicar información de ida y vuelta dentro de un equipo de desarrollo es mediante la conversación cara a cara.*
- *El software que funciona es la principal medida del progreso.*
- *Los procesos ágiles promueven el desarrollo sostenido. Los patrocinadores, desarrolladores y usuarios deben mantener un ritmo constante de forma indefinida.*
- *La atención continua a la excelencia técnica enaltece la agilidad.*
- *La simplicidad como arte de maximizar la cantidad de trabajo que no se hace, es esencial.*
- *Las mejores arquitecturas, requisitos y diseños emergen de equipos que se auto-organizan.*
- *En intervalos regulares, el equipo reflexiona sobre la forma de ser más efectivo y ajusta su conducta en consecuencia.*

Este estilo se tornó muy popular, y hoy son muchos los métodos ágiles de desarrollo de software que se utilizan alrededor del mundo, junto a los cuales se desarrollaron también nuevos métodos de estimación de tamaño funcional, tiempo y costo.

Uno de los métodos más populares es SCRUM, que define los **User Stories** como la forma de describir la funcionalidad del software a desarrollar, el **Product Backlog** como el lugar donde los User Stories son documentados, organizados y priorizados para su desarrollo, y los **Sprints** como los incrementos a realizar en tiempos cortos, en los cuales se busca implementar y entregar al cliente una serie de User Stories funcionando.

Derivado de SCRUM y sus User Stories, nació el método de estimación llamado **Story Points** que se analizará a continuación cómo otra forma posible de proporcionar datos importantes para cálculos y estudios de productividad.

Un Story Point es una representación numérica de la complejidad de un User Story, y del esfuerzo que podría representar para un equipo de desarrollo la implementación de dicho User Story, pero un Story Point comienza a tener sentido recién cuando puede ser analizado en conjunto con otros datos. Por sí mismo un Story Point es solo un número que no dice nada. Otro dato importante es que el Story Point es un valor relativo a la complejidad de otros User Stories seleccionados como referencia. Más adelante se desarrollará esta idea.
Para representar los Story Points se utiliza una escala de números conocida como la escala o la secuencia de Fibonacci, que incluye los números 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 y mayores. En matemáticas, la secuencia de Fibonacci está definida por los 2 primeros números, 0 y 1, y el resto se calcula como la suma de los 2 anteriores en la secuencia. Esta secuencia es muy útil para este tipo de estimaciones ya que los intervalos que genera facilitan la determinación de un valor por ausencia de sus valores vecinos. Por ejemplo, 2 personas estimando no podrán entrar en la discusión fina respecto de si un determinado User Story vale 13 o 14, no existe esa posibilidad.

Otro modo es usar los talles americanos de camisas como XS, S, M, L, XL o XXL, aunque el método con números suele ser más conveniente porque permite fácilmente adicionar Story Points para calcular por ejemplo la cantidad de trabajo realizado en un Sprint.

Figura 2: Story Points y el Planning Poker

Fuente: artículo *Product Management Poker*, pmtribe.wordpress.com, 2010, *p.* 1.

El método de estimación por Story Points contempla el uso de un método de estimación complementario llamado **Planning Poker**. Este método, que es una variante del método Delphi (juicio de experto) analizado antes, contempla el uso de un mazo de cartas que cada participante tendrá, y que utilizará para mostrar su opinión respecto de cuantos Story Points vale el User Story que esté en discusión en ese momento. La estimación se realiza por consenso del equipo de proyecto y es muy importante que participen las personas que finalmente realizarán las tareas (analistas, desarrolladores, testers, DBAs, etc.). El equipo debe trabajar para aunar sus criterios de estimación hasta ponerse de acuerdo respecto de la complejidad de cada User Story y cuantos Story Points representa cada uno. Este trabajo genera mayor compromiso e involucramiento del equipo del proyecto con la estimación y las tareas a realizar.

Es importante también en este método el concepto de la **Velocidad**, que aparece al poder determinar la cantidad de Story Points que fueron implementados en un período de tiempo, por ejemplo en 1 Sprint de 2 semanas. La velocidad se calcula sumando los Story Points estimados para cada User Story que fue implementado en el período, obteniendo un valor que se representa como Story Points por semana o Story Points por mes.

¿Cómo funciona?

Planning Poker es un método de estimación por consenso que se trabaja para lograr la estimación de una lista de funciones (User Stories) que deben ser desarrollados y entregados al cliente. *Planning Poker en Scrum junta las opiniones de varios expertos para la estimación ágil de un proyecto*[16]. Se realiza una reunión de estimación en la que deben participar todos los que van a realizar las tareas de implementación de las funcionalidades, incluida una persona que pueda dar al equipo de desarrollo los detalles de la funcionalidad necesaria (Cliente, Product Owner, etc.).
En esta reunión se sigue este procedimiento:

[16] Definición dada por Mountain Goat Software, una consultora experta en proyectos ágiles, en su web site https://www.mountaingoatsoftware.com//tools/planning-poker

- A cada participante se le entrega el mazo de cartas para la estimación.
- El moderador de la reunión, que no participa de las estimaciones, inicia la discusión sobre una funcionalidad a implementar (User Story).
- Un miembro del equipo con experiencia, alguien del cliente, el product owner o project manager da al equipo todos los detalles posibles sobre el User Story.
- El equipo puede hacer todas las preguntas que necesite sobre el tema y se discute sobre la funcionalidad a implementar.
- Cada persona selecciona la tarjeta que mejor representa su estimación y la coloca boca abajo sobre la mesa para que nadie vea el valor de su estimación.
- Todos los participantes muestran sus tarjetas en forma simultánea. Esto es muy importante para evitar cualquier tipo de influencia de las estimaciones de unos sobre otros.
- Se les pide a los participantes que hayan estimado por debajo o por arriba de la media que compartan sus puntos de vista. Estos pueden explicar y justificar sus estimaciones y la discusión puede continuar por un rato.
- Se repite la votación hasta lograr consenso. El programador con mayor experiencia respecto del tema tratado podría tener doble voto a fin de destrabar una situación en la que no se logra consenso.
- Es posible que el moderador utilice un reloj o timer para estructurar la reunión de forma de no exceder tiempos de discusión lógicos.
- El procedimiento se repite para cada User Story a estimar.
- Al final de la reunión se obtienen una determinada cantidad de User Stories con sus correspondientes Story Points asignados por el equipo de desarrollo.

Habiendo realizado las estimaciones mediante el método Planning Poker, se obtienen como resultado los User Stories estimados (cada uno con sus Story Points asignados) que ahora pueden ser priorizados y comenzar el desarrollo del primer Sprint.

Es importante destacar que en ningún momento se habló de tiempo, ni de esfuerzo (horas hombre), ya que los métodos ágiles definen que no es conveniente intentar determinar al principio del proyecto cuánto tiempo llevará completarlo ni cuánto costará en términos de

esfuerzo. Lo que se debe hacer es iniciar el desarrollo y comenzar a calcular estos valores conforme avance el desarrollo y se terminen los primeros Sprints, y tomando como base del cálculo las estimaciones hechas en Story Points.

Al término de cada ciclo de desarrollo, se podrá calcular cuánto tiempo (días o semanas) y cuánto esfuerzo (horas hombre) fueron necesarios para implementar los Story Points que formaron parte del ciclo. Esto permite comenzar a estimar la velocidad de desarrollo del equipo, y estimar fechas y costos para el resto del proyecto. Estas estimaciones se podrán ir calculando con mayor precisión a medida que se vayan finalizando más Sprints.

A pesar de esta visión de las metodologías ágiles, respecto de que no es aconsejable intentar estimar a largo plazo, entiendo que una organización que realiza sus proyectos con métodos y equipos relativamente estables, podría utilizar información histórica de proyectos anteriores para estimar costos, fechas y estudiar su productividad. Por ejemplo, cada equipo de la organización podría calcular y mantener un índice histórico de productividad calculado en Story Points sobre Horas Hombre. Este índice podría ajustarse con cada proyecto / Sprint y utilizarse para estimaciones y medición de efectividad de mejoras.

Conclusiones del Capítulo:

El cálculo de productividad en desarrollo de software que debe hacerse mediante la relación entre la cantidad del producto construido (tamaño funcional) y el costo incurrido en su producción, según lo visto en este capítulo, podría realizarse utilizando cualquiera de los métodos analizados que permiten determinar el tamaño funcional del producto a desarrollar o ya desarrollado.

LOCs (Lines of Code), FP (Function Points), UCPs (Use Case Points) y SP (Story Points) son dividendos válidos de la fórmula de productividad en desarrollo de software, cuyo divisor será la cantidad de horas-hombre trabajadas, materia prima del proceso:

- P = LOCs / HH
- P = FPs / HH
- P = UCPs / HH
- P = SPs / HH

Cada empresa y/o proyecto de desarrollo de software puede ser medido utilizando el tipo de cálculo de tamaño funcional que mejor se ajuste a su forma de trabajo y metodología, y puede ser estimado en etapas tempranas, se puede ajustar el cálculo con el avance del proyecto, y se puede re-calcular al finalizar para obtener el valor más acertado posible.

El cálculo de productividad realizado a posteriori en cada proyecto permite conocer la productividad real alcanzada en el proyecto y alimentar una base de datos de información histórica que mejore la capacidad de estimaciones de la empresa. El uso de datos históricos de productividad en etapas tempranas de un proyecto, junto con la estimación de su tamaño funcional, permite estimar en forma más precisa los costos y tiempos que el proyecto demandará.

En cualquiera de los casos, la aplicación será beneficiosa si se logra la disciplina organizacional necesaria de seguir los métodos de estimación con rigurosidad, almacenar los datos útiles para futuros proyectos; y si estas actividades se realizan con plena consciencia y responsabilidad de los involucrados respecto de su utilidad y beneficios a futuro.

No hay que olvidar que el software es un intangible, y por lo tanto el cálculo de su tamaño funcional es y será solo una estimación, más o menos precisa, que podría variar si es hecha por diferentes personas que vuelcan al análisis sus diferentes interpretaciones respecto de la complejidad del sistema y sus funciones. Es por eso muy importante que cada empresa, equipo o persona que estima respete las reglas y criterios que cada método define para el conteo.

Medir la productividad permite establecer un baseline o punto de partida, para luego conocer el impacto de las acciones de mejora que se realicen sobre cualquier aspecto del proceso de software, sobre las capacidades de las personas que lo realizan y sobre cualquier aspecto que pueda estar relacionado y/o pueda afectar. Es posible analizar la tendencia de la productividad y poner foco en su mejora, lo que puede resultar en una ventaja competitiva realmente diferenciadora.

Measure to Know

Know to Change

Change to Lead

CAPITULO 3. FACTORES QUE AFECTAN LA PRODUCTIVIDAD

El problema es muy simple visto solamente desde el aspecto matemático. Por tratarse de un simple cociente, se sabe que el resultado de la productividad es claramente afectado por la variación de cualquiera de los términos que intervienen en la división, y en función de cómo se modifica el otro. En los casos más simples, solo uno de los términos se modifica y el otro permanece estable, mientras que en otros casos, ambos términos de la división se modifican generando cambios en la productividad que habrá que estudiar. Se pueden ver y estudiar estos cambios siempre que se hagan en la empresa mediciones periódicas de productividad que permitan comparar períodos, y determinar las variaciones y tendencias del resultado.

Hasta acá, y viendo el problema desde el punto de vista matemático, el tema no revista mayor complejidad. Pero cuando se quieren estudiar las causas de estas variaciones para accionar sobre ellas, el tema se vuelve más complejo. La productividad general, la que se mide en las industrias por ejemplo, suele estar asociada a factores que, en algunos casos, son conocidos, muy estudiados y perfectamente documentados, como por ejemplo las características y especificaciones técnicas de una máquina, o las propiedades de un determinado material que se utiliza como insumo en el proceso productivo en cuestión. Cambiar por una máquina que consume menos energía dará como resultado un aumento de la productividad, si los demás factores no cambian, y esto está especificado y es perfectamente medible y comprobable.

Llevando esto al terreno del desarrollo de software, por ejemplo, si aumenta la cantidad de producto construido con la misma cantidad de horas hombre, se ha logrado un aumento de la productividad, el trabajo del equipo de desarrollo ha sido más productivo. Si por el contrario se ha construido la misma cantidad de producto con más horas hombre, o se ha construido menos funcionalidad con las mismas horas hombre, entonces se ha experimentado una baja en la productividad.

Se explicó en el capítulo anterior cómo se puede medir (al menos por aproximación) el tamaño del producto construido usando métodos de estimación propios de la industria del software como los Use Case Points o los Story Points. El gran desafío ahora es determinar por qué se ha construido menos producto con las mismas horas hombre. ¿Qué es lo que hace que no se pueda garantizar una determinada cantidad de producto a construir con una determinada cantidad de horas hombre? ¿Qué es lo que produce variaciones en la productividad del desarrollo de software? ¿Cuáles son los factores que hay que considerar y sobre los que hay que trabajar para mejorar la productividad?

A lo largo de los años de corta historia que tiene la industria del software, han aparecido diferentes conceptos, metodologías y tecnologías que prometían ser la solución a los problemas de la productividad en desarrollo de software. Todas sin duda aportaron algo, pero ninguna logró dar con la solución definitiva del problema.

En su artículo **No Silver Bullet – Essence and accident in software engineering**, escrito en 1986, Frederick Brooks[17] aporta un concepto interesante al plantear que no habrá solo una forma o método que solucione el problema de una vez y por si solo:

> *There is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement within a decade in productivity, in reliability, in simplicity.*
> *We cannot expect ever to see two-fold gains every two years in software development, like there is in hardware development.*

Frederick Brooks dice que no una, sino una serie se innovaciones que ataquen la complejidad del tema pueden generar una mejora significativa en la productividad del desarrollo de software. La historia demuestra que Frederick Brooks tenía razón, han pasado años, proyectos, métodos, libros y papers, estudios de todo tipo y mucha experiencia

[17] Frederick Phillips Brooks, Jr.: es un ingeniero de software y científico de la computación, nacido en Carolina del Norte (USA) en 1931. Se hizo conocido por dirigir el desarrollo del sistema operativo IBM OS/360 y luego por su libro titulado The Mythical Man-Month (El Mítico Hombre Mes) donde escribe en forma honesta y abierta sobre algunos problemas del desarrollo de software. De este libro sale la famosa Ley de Brooks que dice que "Agregar gente a un proyecto atrasado solo lo atrasará más". También fue famoso y controvertido su paper titulado No Silver Bullet que se menciona en este trabajo.

ganada por la industria, pero hoy los proyectos de desarrollo de software siguen seriamente afectados por problemas de productividad que se reflejan principalmente en atrasos, sobrecostos considerables y problemas de calidad.

Entendiendo que la productividad puede ser afectada por muchos factores, en este capítulo se intentará identificar, agrupar y estudiar aquellos factores que se consideran los principales, los de mayor impacto en la productividad. Identificarlos y estudiarlos dará ideas y algunas respuestas respecto de cómo trabajar con ellos para generar mejoras en la productividad.

En un estudio realizado por Velázquez Rodríguez Bianca Paola[18], titulado simplemente **Desarrollo de Software**, Bianca distingue entre la Productividad Potencial y la Productividad Real de un equipo de desarrollo de software. La productividad potencial es aquella que un equipo puede lograr si hace el mejor uso posible de sus recursos, y la productividad real es aquella que fue efectivamente lograda al final de un proyecto. Lamentablemente, según Bianca, la productividad real, que el individuo o grupo logra, raramente es igual a la potencial ya que los individuos y grupos fallan al hacer el mejor uso posible de los recursos disponibles. Los problemas de comunicación, coordinación y motivación son responsables de insuficiencias en procesos y por consecuencia perdidas en la productividad.

Como parte de un estudio publicado en 1996 por Joseph Blackburn y Gary Scudder[19], ambos profesores de la escuela de Management de la Universidad de Vanderbilt, llamado "**Improving Speed and Productivity of Software Development**"; Blackburn y Scudder condujeron un importante relevamiento que incluyó entrevistas y encuestas a empresas de

[18] Bianca Paola Velázquez Rodríguez: Ingeniera en Computación de la Universidad Nacional Autónoma de México, recibida en Enero del 2013. Publicó durante su carrera varios trabajos cortos sobre temas relacionados a desarrollo de software, e incluso su tesis final titulada "Sistema Web de Material de Apoyo a la Docencia para la División de Ciencias Sociales y Humanidades de la F.I.".

[19] Joseph Blackburn y Gary Scudder, ambos profesores de Management de la Universidad de Vanderbilt, junto con Luk Van Wassenhove de la empresa Insead de Francia, publicaron este trabajo en 1996, basado en encuestas, donde buscan entender el impacto de las acciones de gestión sobre la productividad y velocidad del desarrollo de software.

desarrollo de software de Estados Unidos, Europa y Japón. Los datos analizados y correlacionados les permitieron llegar a las siguientes conclusiones:

- Mejorar el tiempo total del proyecto no implica reducir el tiempo de todas las etapas, sino, por el contrario, dedicar más tiempo a algunas, especialmente las iniciales. Llaman a esto The Time Paradox.

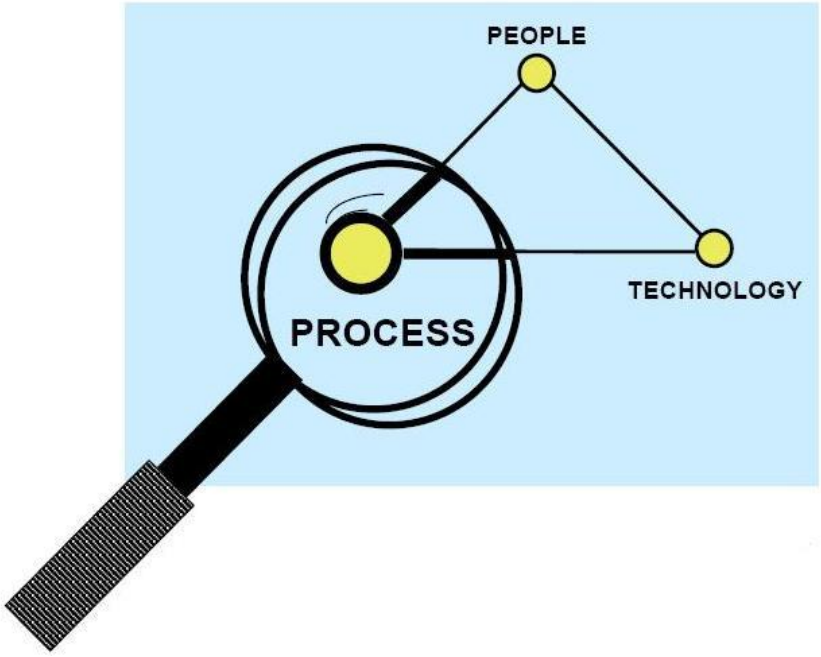
- Small is Better. Las empresas más productivas tienen equipos de desarrollo pequeños, los cuales parecen ser más productivos que los grandes equipos que en general padecen de serios problemas de comunicación y coordinación. Ya lo decía F. Brooks, el autor de No Silver Bullet, en sus estudios cuando mencionaba que en desarrollo de software "more is less". Blackburn y Scudder llaman a esto The Productivity Paradox.

- People and Process Matter, Not Tools. El estudio de Blackburn y Scudder, refleja la opinión de muchos managers de software y muestra la correlación de los datos obtenidos que parecen dejar en claro que el factor más importante son las personas y su experiencia. La correlación positiva más importante de las variables de tiempo y productividad, se da con las que indican la experiencia de los equipos de trabajo.

Blackburn y Scudder concuerdan con F. Brooks, No hay balas de plata cuando se trata de solucionar los problemas de productividad en desarrollo de software, pero son categóricos al mencionar que "son las personas las que hacen la diferencia, porque son ellos los que deben ejecutar el plan y en una actividad creativa como es el diseño de software, no hay sustituto para el talento".

Por otro lado, el SEI, Software Engineering Institute, de la Universidad Carnegie Mellon, es el responsable del desarrollo del modelo SEI CMMi, uno de los modelos más usados del mundo para el desarrollo y mejora de procesos de software.

Figura 3: La visión del SEI - Foco en los Procesos



Fuente: SEI-CMU, *CMMi Version 1.2 Overview,* USA, 2007, *p.* 7.

El SEI presenta un enfoque diferente al explicar el motivo por el cual pone foco en el desarrollo y la madurez de los procesos, ellos mencionan que "La calidad de un producto o servicio está muy influenciada por la calidad del proceso utilizado para desarrollarlo y mantenerlo". Este foco se explica con el siguiente argumento: Las empresas de desarrollo de software, sin importar su nivel de madurez, tienen sin excepción, personas en su staff que utilizan herramientas tecnológicas para el desarrollo de software. Un buen proceso les indicará a las personas, cómo deben usar la tecnología disponible para lograr los resultados esperados en los proyectos de desarrollo de software. Este enfoque del SEI coincide con la conclusión de Velázquez Rodríguez Bianca Paola que menciona que "los individuos y grupos fallan al hacer el mejor uso posible de los recursos disponibles".

Los estudios analizados hasta ahora parecen indicar que las personas con su experiencia, nivel de entrenamiento y motivación, junto a procesos y métodos adecuados pueden ser factores diferenciales para la productividad en esta actividad.

Stefan Wagner del Instituto de Informática de Múnich, y Melanie Ruhe de Siemens AG, publicaron en Septiembre del 2008 un estudio titulado **A Structured Review of Productivity Factors in Software Development**[20]. En este estudio, Stefan y Melanie realizan una revisión detallada de diferentes trabajos publicados entre 1970 y 2007 en los que se intenta determinar y analizar los factores que influyen en la productividad de las actividades de desarrollo de software. El objetivo del trabajo de Stefan y Melanie es llegar a construir una única lista de factores que contenga todos los que han sido identificados en trabajos anteriores, que al 2008 continúan siendo relevantes, y que pueda ser usada como base para futuros análisis y para construir modelos de productividad.

El trabajo dio como resultado una lista de 51 factores clasificados de la siguiente manera:

Cuadro 11: Clasificación de Factores de S. Wagner y M. Ruhe

• **Factores Técnicos (Hard):** o Del Producto (12) o Del Proceso (8) o Del Entorno de Desarrollo (4)	• **Factores Humanos (Soft):** o Cultura Corporativa (3) o Cultura del Equipo (8) o Capacidades y Experiencia (8) o Entorno – Ambiente Físico (5) o Del Proyecto (3)

Fuente: S. Wagner y M. Ruhe, *A Structured Review of Productivity Factors in Software Development*, Munich, 2008, *p*. 12-15.

Una de las cuestiones en las que quisieron hacer énfasis en este trabajo es el hecho de que la mayoría de los estudios sobre los factores que afectan la productividad en desarrollo de software se centraban en los factores técnicos (o hard), dejando de lado los factores humanos. Es por eso que pusieron especial énfasis en encontrar y analizar trabajos que consideren también los factores humanos que aquí denominaron "**Factores Soft**".

La lista completa de factores identificados y clasificados por Stefan y Melanie se publica a continuación:

[20] Este estudio también puede ser encontrado con el título **A Systematic Review of Productivity Factors in Software Development**.

Cuadro 12: Lista de Factores de S. Wagner y M. Ruhe

Factores Técnicos (Hard)	Factores Humanos (Soft)
del Producto	**Cultura Corporativa**
Similitud con otros proyectos.	Credibilidad
Confiabilidad Requerida del Software	Respeto
Database Size	Reconocimiento
Complejidad del Producto	**Cultura de Equipo**
Necesidad de Reusabilidad	Camaradería
Restricciones de Tiempo del Proyecto	Identidad del equipo
Restricciones de Almacenamiento	Sentido de Elite
Tamaño del product	Objetivos Claros
Calidad del Producto	Rotación
Complejidad de la interface de usuario	Cohesión
Flexibilidad de las Restricciones	Comunicación
Reúso	Soporte para la Innovación
del Proceso	**Capacidades y Experiencia**
Riesgos Mitigados en la arquitectura	Temperamentos de los miembros
Madurez del Proceso	Capacidad de Análisis
Volatilidad de la Plataforma	Capacidad de Programación
Prototipación Temprana	Experiencia en la aplicación
Completitud del Diseño	Experiencia con la plataformas (SO / HW)
Efectividad de la Detección de Defectos	Experiencia con el lenguaje y las herramientas
Duración del Proyecto	Capacidades del manager
Concurrencia con el desarrollo del Hardware	Experiencia del Manager con la aplicación
del Entorno de Desarrollo	**Entorno - Ambiente Físico**
Uso de Herramientas de Software	Espacio de Trabajo
Lenguaje de Programación utilizado	Nivel de Interrupciones
Uso de Prácticas modernas de Desarrollo	Fragmentación del Tiempo
Documentación ajustada a las necesidades	Separación Física
	Facilidades de Telecomunicaciones y Teletrabajo
	Proyecto
	Compromisos de Fechas
	Estabilidad de los Requerimientos
	Tamaño del equipo

Fuente: S. Wagner y M. Ruhe, *A Structured Review of Productivity Factors in Software Development* , Munich, 2008, *p*. 12-15.

Con esta misma línea de pensamiento, Tyson R. Henry[21] profesor de la California State University, Chico; publicó en 2005 un trabajo titulado "**Software Development Productivity: Considering the Socio-Technical Side of the Software Development**". En su estudio Tyson Henry dice que la promesa implícita de la ingeniería de software de que un proceso sistemático y bien desarrollado permite a cualquier grupo desarrollar cualquier software de manera eficiente, claramente no es verdad. Además de procesos, herramientas y técnicas, es necesario que la ingeniería de software ponga atención en la foto completa, incluyendo el estudio del lado humano de la actividad; lo que él llama la Ingeniería de Software Socio-Técnica. El potencial de la ingeniería de software socio-técnica, como la llama Tyson Henry, es tremendo: los procesos de ingeniería de software podrían ser modificados para ser más ampliamente aceptados por las personas y efectivos; herramientas, reglas de negocio, estructuras organizacionales, formas de comunicación y métodos de management podrían ser revisados y mejorados con criterios que contemplen el impacto en las personas y su motivación.

Algunos ejemplos citados en Tyson Henry que sirven para ilustrar el impacto de los factores sociales o humanos en la ingeniería de software:

- Linberg, Procaccino y Verner demostraron en estudios con simples cuestionarios realizados a grupos de desarrolladores, cómo ellos tienen un entendimiento de lo que significa el éxito del proyecto totalmente distinto al entendimiento del mismo concepto por parte de la organización, y cómo la idea de satisfacción con el trabajo por parte de los desarrolladores estaba totalmente alejada del cumplimiento de los objetivos del negocio. La percepción y entendimiento de éxito y satisfacción con el trabajo de los desarrolladores suele estar asociada a factores de satisfacción personal que no contemplaban los objetivos reales del negocio, evidenciando un problema de comunicación.

[21] Henry R. Tyson: PhD. en Ciencias de la Computación y Profesor Asociado del Departamento de Ciencias de la Computación de la California State University, Chico. Es profesor de esta universidad desde 2001, pasando antes por empresas como Sandia National Laboratories, Nicholes Research Corporation y Applied Research Associates. Es autor de varias publicaciones en el campo del desarrollo de software.

- Otro estudio mencionado por Tyson R. Henry sobre las inspecciones realizadas en una organización (revisiones de código, por ejemplo), indicó que las inspecciones realizadas por personas cercanas a los autores del código (cercanas en términos de la organización, o por amistad personal) tienden a dedicar menos tiempo a discutir sobre los hallazgos y tienden a reportar menos fallas. Un aspecto socio-técnico poco tenido en cuenta que podría afectar la calidad del software y en consecuencia también la productividad de manera visible.

Según el estudio de Tyson Henry, hay una incontable cantidad de aspectos a estudiar por parte de la ingeniería de software socio-técnica y no es posible hoy determinar con precisión cuáles son los aspectos más importantes para ser estudiados primero; pero un análisis preliminar de los trabajos existentes muestra que las áreas o aspectos al menos más referenciados hoy son: la colaboración y trabajo en equipo, la comunicación, las prácticas o métodos de management y el entorno de oficina o entorno físico.

En **Understanding Software Productivity**, un trabajo realizado en 1994 en la Escuela de Administración de Negocios de la Universidad de Southern California, su autor Walt Scacchi[22] identifica los atributos de un proyecto de software que según él facilitan alcanzar una alta productividad:

Atributos del Ambiente donde se desarrolla Software:

- Actividades de Desarrollo rápidas y gran capacidad de procesamiento.
- Infraestructura de computación y recursos de Soporte abundantes.
- Técnicas y herramientas de ingeniería de software modernas.
- Ayudas tecnológicas para soportar desarrollos de gran escala (herramientas de configuración, diseño, comunicación, etc.).
- Lenguajes de muy alto nivel, cercanos al lenguaje del dominio.

[22] Walt Scacchi: Investigador Científico Senior en el Institute for Software Research, y Director de Investigación en el Institute for Virtual Environments and Computer Games, ambos en la Universidad de California, Irvine. Recibió su Doctorado en Ciencias de la Información y la Computación en 1981. Entre 1981 y 1998 fue profesor en la Universidad de Southern California, regresando a Irvine en 1999.

- Entornos de desarrollo centrados en procesos que coordinen el trabajo de muchos grupos pequeños.

Atributos del producto de Software:

- Desarrollar sistemas de complejidad pequeña o media.
- Reusar software que Soporte el procesamiento de información requerido.
- No desarrollar sistemas de tiempo real ni sistemas distribuidos.
- Mínimas restricciones a la seguridad o integridad de los datos.
- Requerimientos y Especificaciones estables.
- Proyectos cortos para minimizar la posibilidad de cambios.

Atributos del Staff:

- Equipos experimentados, pequeños y bien organizados.
- Staff de desarrollo experimentado en el dominio y/o proyectos similares.
- Desarrolladores y managers que colecten y evalúen sus propios datos de productividad y sean premiados por esto.
- Varias estructuras de equipos y la capacidad de cambiar entre ellas confirme avanza el trabajo.

Es posible comenzar a identificar las primeras coincidencias entre los factores que diferentes autores han identificado como los que pueden afectar la productividad:

- Los equipos fallan en alcanzar su productividad potencial según Velázquez Rodríguez Bianca Paola.
- “Las personas hacen la diferencia” según Blackburn y Scudder y debería dedicarse más tiempo a ciertas actividades.
- El SEI pone foco en la madurez de los procesos como guía para las personas.
- No se presta suficiente atención a los aspectos humanos de la ingeniería de software según Megan y Ruhe.
- Tyson R. Henry habla de la ingeniería de software socio-técnica.

- Walt Scacchi identifica varios atributos del staff como la experiencia y los equipos pequeños y dinámicos.

El hecho de ser la ingeniería de software una actividad básicamente humana, basada en el conocimiento, la creatividad y el talento, hace pensar que el factor que produce sus mayores éxitos y sus mejores resultados, es también aquel que genera las mayores complicaciones, atrasos y desvíos en los proyectos. Las personas.

Muchos estudios muestran la variabilidad en la productividad que se produce cuando una misma actividad es realizada por diferentes personas, producto de sus diferentes habilidades. Bill Curtis[23] en **Substantiating Programmer Variability, Proc. IEEE 1981** dice que las variaciones de productividad entre los individuos pueden ser de más de un orden de magnitud. Esta variación suele quedar disimulada en proyectos o equipos muy grandes donde la productividad del programador promedio es la que domina y esconde las variaciones individuales, pero suele ser diferente en proyectos o equipos chicos donde las diferencias individuales dominan y pueden hacer la diferencia. Esta misma variabilidad que demuestra Curtis sobre los programadores puede ser llevada a todos los roles y funciones de la ingeniería de software con el impacto que cada una podría tener sobre la productividad, desde un error menor (de código) generado por un programador con poco entrenamiento, hasta un error de concepto grave, introducido por un diseñador en las etapas tempranas de un proyecto.

La literatura revisada, las respuestas recibidas de profesionales y colegas consultados y la propia experiencia profesional del autor de este trabajo en la industria del software permiten identificar y clasificar los factores que influyen en la productividad del desarrollo de software de la siguiente manera:

23 Bill Curtis: nació en Estados Unidos en 1948 y desarrolló una brillante carrera como especialista en ingeniería de software y temas organizacionales. La etapa más conocida de su trabajo es cuando lideró el desarrollo de los modelos Software CMM y People CMM en el SEI Software Engineering Institute de la Universidad Carnegie Mellon. El su estudio titulado Substantiating Programmer Variability, Bill Curtis afirma que hay una gran oportunidad para mejorar la productividad si se investiga y se trabaja para reducir la gran variabilidad que hay en la performance de los programadores.

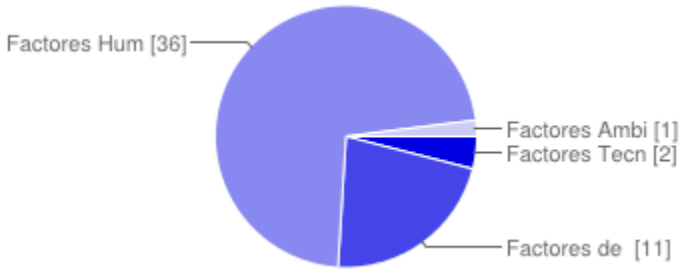
- **Factores Humanos**: todo lo relacionado a las personas, especialmente la experiencia, el nivel de entrenamiento del equipo, y la motivación.
- **Factores de Proceso**: cuestiones metodológicas, procesos utilizados, formas de trabajo, definición de roles y responsabilidades, coordinación y comunicación (o la ausencia de todos ellos).
- **Factores Tecnológicos**: equipamiento, comunicaciones, herramientas de desarrollo, lenguajes y compiladores, herramientas de ayuda al diseño, equipos y capacidad de cómputo dedicada al desarrollo, etc.
- **Factores Ambientales**: características físicas de los lugares donde se llevan a cabo las actividades de desarrollo de software.

Sobre este primer grupo de aspectos que afectan la productividad, el grupo de los aspectos humanos, la experiencia indica que es posible distinguir entre 2 aspectos clave: el entrenamiento y la motivación. Por el hecho de que una persona, para realizar una determinada actividad de manera correcta y eficiente debe poder hacerla (capacitación) y debe querer hacerla (motivación).

Las encuestas realizadas como parte de este trabajo de investigación respaldan esta afirmación ya que la mayoría de los encuestados coinciden en destacar que los aspectos humanos son los principales que afectan la productividad en desarrollo de software, por sobre los aspectos técnicos, metodológicos y ambientales. Los aspectos metodológicos o de procesos con considerados por la mayoría en segundo lugar:

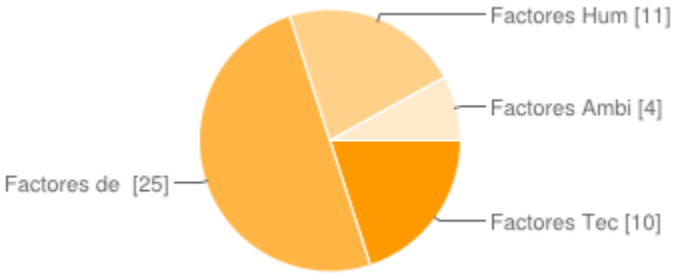
Datos tomados de la encuesta realizada en Abril 2014 a profesionales de la industria del software de la República Argentina. Sobre la base de 50 respuestas, se obtuvieron los siguientes resultados para las preguntas referidas a factores que afectan la productividad:

Si agrupamos los factores que pueden aumentar o reducir la productividad en desarrollo de software en los 4 grupos abajo listados; ¿cuál te parece que es el grupo más relevante?



Factores Humanos (entrenamiento, motivación, etc.).	**36**	**72%**
Factores de Procesos (metodologías, estándares, etc.).	**11**	**22%**
Factores Tecnológicos (equipos, herramientas).	**2**	**4%**
Factores Ambientales (oficinas, espacio, luz, ruido, etc.).	**1**	**2%**

¿Y cuál te parece que es el segundo grupo en relevancia?



Factores de Procesos (metodologías, estándares, etc.).	**25**	**50%**
Factores Humanos (entrenamiento, motivación, etc.).	**11**	**22%**
Factores Tecnológicos (equipos, herramientas).	**10**	**20%**
Factores Ambientales (oficinas, espacio, luz, ruido, etc.).	**4**	**8%**

Las respuestas indican que los Factores Humanos son los que más afectan la productividad para el 72% de los profesionales que respondieron la encuesta, seguidos por los Factores de Procesos para el 22% de la muestra. En la segunda pregunta, realizada con el fin de constatar el segundo factor, se confirma que los Factores de Proceso son los que le siguen a

los Factores Humanos. Luego, los Factores Tecnológicos en tercer lugar y los Factores Ambientales parecen ser los que menos impactan.

Otra consideración interesante es que el 100% de los Owners y Top Managers encuestados opinan que son los Factores Humanos los más importantes; mientras que en el caso de los más experimentados, los que tienen más de 12 años de experiencia en la industria, el 73% considera a los Factores Humanos en primer lugar y a los Factores de Proceso en segundo lugar. Los otros 2 grupos de factores no son mencionados por los más experimentados.

Como conclusión final de este capítulo, se entiende que los factores deben ser considerados en el orden mencionado antes: Factores Humanos, Factores de Procesos, Factores Tecnológicos y Factores Ambientales.

Por ser las personas las protagonistas de la actividad, es lógico y razonable que los Factores Humanos sean los de mayor impacto, ya que *desarrollar software es transformar conocimiento*[24]. No hay máquinas o insumos que puedan ser reemplazadas o mejoradas para lograr un aumento importante de la productividad. Indefectiblemente el foco debe estar puesto en trabajar con y para el motor principal de esta actividad: las personas.

El gran desafío que esto plantea es que los factores más importantes son a su vez los más complejos de entender y tratar para lograr un cambio y una mejora. No basta con incorporar mejores herramientas y tecnologías, equipos más potentes u oficinas más modernas; primero es necesario trabajar para que las personas que integran los equipos de desarrollo de software se sientan motivadas y sean capaces de dar lo mejor de sí mismas para el éxito de los proyectos, éxito que a su vez debería coincidir con lo que cada persona busca en sus aspiraciones y sus deseos de realización personal y profesional[25].

[24] Concepto tomado de la entrevista realizada a un referente de la industria del software en Argentina. Ver Anexo 3.

[25] Tyson R. Henry nos habla de la ingeniería de software socio-técnica en *Software Development Productivity: Considering the Socio-Technical Side of the Software Development* y menciona las diferencias que se presentan en lo que las personas y las empresas entienden y perciben como éxito de los proyectos.

CAPITULO 4. RELACION ENTRE PRODUCTIVIDAD Y CALIDAD

Según demuestran muchos estudios de la industria del software, el costo de detectar y remover las fallas de un software, en muchos casos es más alto que el mismo costo de haber construido el sistema. Hay sistemas que luego de periodos relativamente cortos de diseño y construcción, por ejemplo 1 año, pasan por un periodo mucho más largo en uso y mantenimiento, donde se insumen muchas horas hombre para detectar y solucionar las fallas que no fueron detectadas durante la construcción y el período inicial de pruebas. A veces son muchas más horas que las que se usaron para el desarrollo original del sistema.

La actividad de desarrollar software, por ser una actividad humana, es propensa a generar errores en todas las etapas del proceso. Las personas se equivocan, cometen errores, es parte de la naturaleza humana, y por consiguiente es parte también de la naturaleza del proceso de desarrollar software.

Las fallas en el software son algo absolutamente natural y hay que aprender a gestionarlas de la mejor manera para que su impacto en el proceso sea mínimo. No es posible pensar que no estarán, no es posible o es inocente pensar en un proceso de desarrollo de software en el cual no se generarán fallas. Por el contrario, hay que asumir que la generación de fallas es natural, es parte del proceso y por eso es muy importante gestionarlas para detectarlas y eliminarlas.

El problema con las fallas en el software, no reside en el hecho de que se presenten, porque como se dijo antes, es natural que así ocurra. El problema se presenta cuando las fallas que se generan no son detectadas y eliminadas lo más rápido posible; y se las deja avanzar hasta cuando su detección y solución tiene un impacto alto en el proyecto, en tiempo, costo y satisfacción del cliente.

Cada falla que se genera durante el desarrollo de un software, representa un costo adicional, generalmente materializado en horas hombre y tiempo, lo que en definitiva es dinero. Este

sobre costo será cada vez mayor conforme la falla no sea detectada y pase a etapas posteriores del proceso. Será el caso más extremo el de una falla que se genera en la primera etapa de relevamiento de los requerimientos de los usuarios, un error conceptual respecto de una funcionalidad del sistema, y esta falla no es detectada en esta misma etapa ni en ninguna de las posteriores, dando como resultado un software desarrollado y probado sobre la base de un error de concepto. Recién al final, cuando el sistema es puesto en servicio, el usuario se da cuenta de este error, dando origen a una gran cantidad de re-trabajo para modificar el requerimiento fallado y a partir de allí todas las especificaciones y el software propiamente dicho que deberá modificado y nuevamente testeado y entregado al usuario.

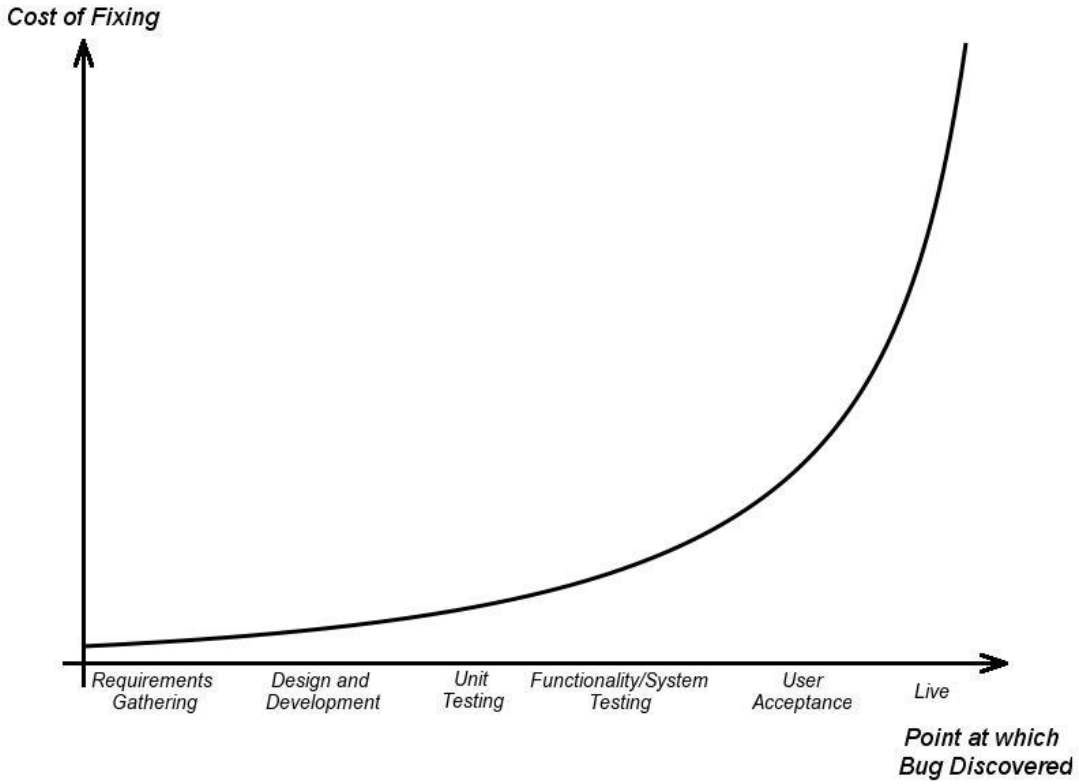
Se perdió mucho tiempo y dinero y el costo adicional difícil de medir de la percepción del cliente o usuario de que el sistema no funciona y no hace lo que el necesita. En el otro extremo, si la misma falla hubiera sido detectada en la misma etapa donde se genero, mediante alguna actividad de revisión de los requerimientos, o una presentación a los usuarios de cómo fueron interpretadas sus necesidades, esa misma falla se hubiera detectado y solucionado con un impacto mínimo y un sobre costo que podría haber sido despreciable.

Hay estudios que indican que el costo de remover una falla de software aumenta en forma exponencial conforme avanza el proyecto[26], siendo una falla que se genera al inicio del proyecto muy barata de solucionar al inicio del proyecto si es detectada en forma temprana, y muy costosa de solucionar sobre las etapas finales del proyecto o la etapa de producción si recién allí es detectada.

De lo expuesto anteriormente es posible deducir que hay una relación muy importante que es posible establecer entre la productividad de un equipo de desarrollo, la cantidad de fallas que este genera y la forma en que las gestiona en su proceso de desarrollo de software.

[26] Estudios del IBM Systems Sciences Institute demuestran que una falla detectada en producción puede costar hasta 100 veces más que la misma falla detectada en la etapa de diseño. Publicado en http://www.isixsigma.com/

Figura 4: Costo de las Fallas.



Fuente: IBM Systems Sciences Institute.

Si se considera como Calidad al tratamiento que un equipo de desarrollo hace en su proceso para detectar y solucionar fallas en forma temprana, es posible afirmar que un equipo de desarrollo que trabaja con un alto nivel de calidad es aquel que detecta sus fallas y las soluciona en el punto más bajo posible de la curva de costo de las mismas. Las detecta y las soluciona lo más rápido posible luego de haberlas generado. Por otro lado, un equipo que trabaja con bajos niveles de calidad generará fallas que no serán detectadas sino hasta etapas avanzadas del proyecto donde el costo de solucionarlas será mucho más alto.

La productividad de ambos equipos será diferente ya que uno puede construir la funcionalidad solicitada con la cantidad de horas hombre que sean necesarias y poco impacto de re-trabajo producto de las fallas; mientras que el otro usará muchas más horas hombre para construir la misma funcionalidad producto de las fallas que deberá solucionar sobre el final del proceso.

Entonces, la relación entre Productividad y Calidad está dada de forma que cuanto mayor sea la atención del equipo de desarrollo en la gestión de sus fallas (foco en la Calidad) mejor será su productividad ya que usará menos horas hombre para construir el software.

Este concepto se ve claramente con un ejemplo simple: 2 equipos de desarrollo de software realizarán un desarrollo que es similar en complejidad y en tamaño funcional, estimado en 280 UCPs (Use Case Points). El equipo AC (Alta Calidad) pone mucho foco en la calidad y su proceso de desarrollo le permite detectar y solucionar las fallas con mínimo impacto. El equipo BC (Baja Calidad) usa un proceso similar pero no pone tanto foco en la detección temprana de las fallas, de hecho su historia muestra que es común en ellos que las fallas se detecten avanzado el proyecto. Ambos equipos realizan el desarrollo y los números resultantes son:

Cuadro 13: Ejemplo - Comparación de productividad de 2 equipos

	Equipo AC	**Equipo BC**
Tamaño funcional medido en UCPs	280	280
Horas Hombre Desarrollo	3000	3000
Horas Hombre solución de fallas	150	1500
Productividad (UCPs por cada 100 HH)	**8,89**	**6,22**

Fuente: Ejemplo de elaboración propia.

El bajo impacto de las fallas que logra el equipo AC le permite lograr un mejor valor de productividad, que resulta ser un 50% mayor que la productividad del equipo BC.

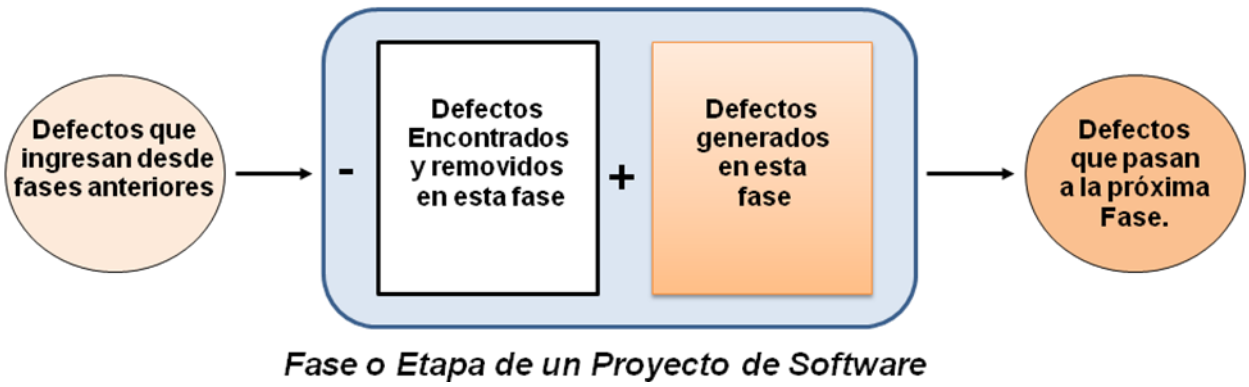
Pasa un tiempo y la misma empresa debe contratar un equipo de desarrollo para realizar la construcción de un sistema mucho más grande, estimado en 2500 UCPs. Los datos del desarrollo anterior permiten estimar que el equipo AC realizará el desarrollo con unas 28.000 horas hombre, mientras que el equipo BC lo podrá realizar con 40.000 horas hombre. ¿A quién cree usted que la empresa va a contratar?

Muchas fallas detectadas tarde en el proceso darán como resultado muchas más horas-hombre necesarias para generar la misma cantidad de funcionalidad con un nivel de calidad aceptable que permita la operación del software en producción y con un nivel de aceptación adecuado por parte de los usuarios. Claramente esto reduce la productividad.

Se trata entonces de detectar las fallas lo antes posible para no dejarlas pasar a las próximas etapas cuando su resolución es más costosa. Existe para eso una medición llamada DRE (Defect Removal Efficiency) que permite estudiar el proceso de desarrollo de software de una empresa para determinar cuán eficiente es para detectar y remover las fallas[27].

La medición DRE se realiza a través de la clasificación de las fallas que se generan, registrando para cada falla cual fue la etapa del proceso en la cual la falla fue detectada, y cuál fue la etapa que inyectó la falla. Esto requiere algo de trabajo adicional en registración y en la clasificación de las fallas, especialmente para determinar el origen (etapa) de la falla. Los datos de las fallas de uno o varios proyectos pueden ser luego volcados a la matriz DRE que permite visualizar cuantas fallas se inyectaron en cada etapa, cuantas fallas fueron detectadas en cada etapa, y uno de los más importantes, cuantas fallas cada etapa dejo pasar sin haberlas detectado.

Figura 5: esquema de un análisis DRE



Fuente: elaboración propia.

[27] Caper Jones define la medición DRE como "*the ratio of bugs found prior to installation of a software application to the total number of bugs in the application*", Applied Software Measurement, 2008 (Kindle Locations 1691-1692).

Por ejemplo, en un proceso de software básico, tipo Waterfall, de 6 etapas: requerimientos, análisis, diseño, codificación, testing, despliegue; para una falla generada en requerimientos y removida en testing, la matriz DRE contabiliza:

- La falla fue inyectada y no detectada en Requerimientos.
- La falla fue detectada y removida en Testing.
- Las falla paso sin ser detectada ni removida por análisis, diseño y codificación.

Los datos del origen y muerte de la falla, en realidad no son los más importantes en sí mismos, sino que permiten determinar con algunas cuentas básicas que etapas del proceso no son buenas detectoras de fallas, o lo que es lo mismo, tienen una baja efectividad de remoción de defectos (DRE).

La matriz conoce cuantas fallas ingresaron a una etapa generadas por etapas anteriores, cuantas fallas fueron generadas en esa misma etapa y cuantas fallas fueron detectadas en esa etapa. Esto permite determinar rápidamente la efectividad de remoción de defectos de esa etapa.

Ejemplo, datos para la etapa de diseño:

- Fallas que ingresaron de etapas anteriores (FA): 12
- Fallas inyectadas en la propia etapa (FP): 4
- Fallas removidas en la etapa (FR): 8
- DRE de la Etapa = FR / (FA + FP) = 8 / (12 + 4) = 8 / 16 = 50%.

En el ejemplo anterior, la etapa de Diseño tuvo una efectividad de remoción de defectos del 50% simplemente porque en ella se detectaron y removieron la mitad de las fallas que estaban presentes en la etapa. La otra mitad pasó sin ser detectada a etapas posteriores del proyecto donde su detección y remoción ya es más costosa. Este valor es indicador de que es necesario hacer algo para mejorar la efectividad de las actividades de inspección o revisión de esta etapa.

El valor esperado para esta medición sería del 100% lo que significa que en cada etapa se detectan y eliminan todas las fallas existentes y que no hay fallas que pasan de una etapa a otra. Es cierto que valores del 100% son muy difíciles de alcanzar y mantener, y principalmente muy costosos; pero se debería intentar que la medición se aproxime lo más posible al 100%.

Ese es el objetivo, no se trata de no generar fallas, porque como se mencionó antes, generar fallas es natural y contra eso no es posible luchar. Se trata de detectar una falla lo antes posible, antes de que avance en el proceso haciendo cada vez más costosa su resolución. Para esto, las etapas del proceso de software pueden ser complementadas con actividades de revisión, inspección, presentaciones, reuniones de discusión, prototipos, tendientes a detectar y solucionar fallas antes de continuar.

La matriz DRE muestra un mapa respecto de la detección de los defectos, cuales etapas son buenas detectoras de defectos y cuales los dejan pasar; también muestra cuales etapas son las que inyectan más defectos. Con este mapa es posible tomar decisiones respecto de en cuales etapas hay que mejorar las actividades del proceso de software para detectar y remover fallas, y permite además, medir fácilmente la efectividad de los cambios que se hagan.

La medición DRE puede ser usada tanto para procesos de software clásicos del tipo Waterfall y RUP, como para procesos ágiles como SCRUM. La principal diferencia es que en los procesos clásicos es posible realizar un estudio más granular al nivel de cada etapa del proceso, mientras que para SCRUM solo se puede identificar y medir la etapa del desarrollo respecto de las fallas que se entregaron en el release de cada Sprint.

La debilidad de este método reside en que el mapa termina de construirse recién al final. Es necesario esperar al fin del proyecto y luego de un tiempo transcurrido con el sistema en funcionamiento para asegurar que todas, o al menos la mayoría de las fallas, fueron detectadas y contabilizadas.

Una mejora en la capacidad de nuestro proceso de desarrollo de software de detectar y remover en forma rápida las fallas que se generen dará como resultado una mejora visible e importante en la productividad del equipo de desarrollo.

Cierra este capítulo con una nueva cita textual del artículo **No Silver Bullet – Essence and accident in software engineering**, escrito en 1986 por Frederick Brooks:

> "*La parte dura de hacer software es la especificación, diseño y testing de la construcción conceptual, no el trabajo de representarlo y testear la fidelidad de la representación. Todavía cometemos errores de sintaxis, seguramente, pero estos no son nada comparados con los errores conceptuales en la mayoría de los sistemas*".

CAPITULO 5. PRODUCTIVIDAD EN TERMINOS DEL NEGOCIO

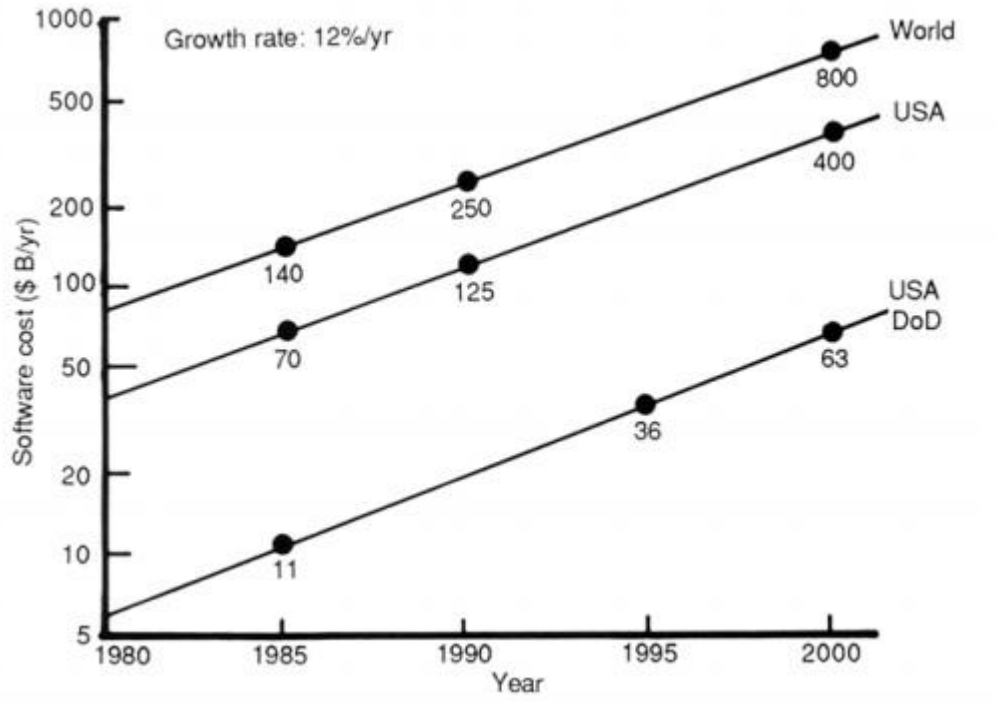
No es de menor importancia la relación que las actividades de desarrollo de software tienen con variables del negocio tan básicas como son el costo y la rentabilidad; especialmente en empresas que se dedican a desarrollar software y es este su negocio principal.

Llevando el tema del costo del software a nivel mundial y para apreciar la magnitud económica que podría tener una mejora en la productividad, se toman los datos publicados por Barry W. Boehm[28] en 1987 en su artículo titulado **Improving Software Productivity**. Barry Boehm opina que la mayor motivación para lograr una mejora en la productividad es que el costo del software es muy alto, y su ratio de crecimiento anual es muy alto también. Según reporta Boehm, el costo del software en 1985 solo para los Estados Unidos fue de U$S 70 billones, y de U$S 140 billones en todo el mundo. También indica que en ese momento el costo del software crecía a una tasa del 12% anual. Por eso en su artículo, Boehm calcula que si ese ratio de crecimiento se mantiene, el costo del software en 1995 será de U$S 225 billones solo en Estados Unidos y de U$S 450 billones en el mundo entero. Una mejora de la productividad del orden del 20% en aquel momento hubiera implicado un ahorro de U$S 45 billones en Estados Unidos y U$S 90 billones en el mundo.

Continuando con la proyección de Boehm, y asumiendo un crecimiento anual y constante del 12%, en 2010 el costo del software para los Estados Unidos hubiese sido del orden de los U$S 1190 billones, y del orden de los U$S 2380 billones para el mundo entero. En este caso, una mejora de la productividad del 20% hubiese generado ahorros de U$S 238 billones y U$S 476 billones para USA y el mundo entero respectivamente.

[28] Barry W. Boehm, es un ingeniero estadounidense nacido en 1935 que realizó importantes aportes al campo de la ingeniería de software, entre ellos el métodos de estimaciones COCOMO, el método espiral de desarrollo de software y varias publicaciones. Luego de su paso por varias compañías de software, y por el Departamento de Defensa de Estados Unidos, desde 1992 es profesor de ingeniería informática en el departamento de ciencias de la informática en la Universidad del Sur de California.

Figura 6: Costo del Software entre 1980 y 2000.



Fuente: Barry Boehm, *Improving Software Productivity*, USA, 1987, *p.* 2.

Dado el crecimiento explosivo que experimentó la industria del software en los últimos 15 años de la mano de internet y los dispositivos móviles, es posible asumir que los números reales superaron bastante a las proyecciones de Boehm, con lo cual las cifras reales son aún más impresionantes.

Volviendo ahora al ámbito de las empresas, todo proyecto de software es llevado a cabo a riesgo de no lograr un producto terminado y entregable al cliente dentro de los plazos estimados y principalmente dentro de los costos estimados. Si esto ocurre, el aumento de horas hombre respecto de la estimación original comenzará a erosionar el margen de rentabilidad del proyecto corriendo el riesgo de llevarlo al extremo de terminar haciéndolo a pérdida o tener que cancelarlo.

Las horas hombre en las actividades de desarrollo de software, suelen ser el principal componente del costo de los proyectos; y es por eso sumamente importante que los niveles de productividad obtenidos permitan que este costo se mantenga dentro de las estimaciones realizadas. La relación que existe entre la productividad y el costo es simple y directa:

- Si la productividad no es buena aumentará la cantidad de horas hombre necesarias para finalizar el desarrollo.
- Si aumenta la cantidad de horas hombre, aumenta también el costo del proyecto, producto del aumento de su componente principal.
- Si aumenta el costo, se reducirá la rentabilidad del proyecto, generando en consecuencia una disminución de la rentabilidad general de la empresa o de la unidad de negocio.

Por supuesto que esto depende del modelo de contratación que se esté utilizando. En los modelos de contratación a **precio fijo** por un producto terminado (lo que comúnmente en el mercado argentino se denomina Llave en Mano), la empresa que desarrolla el software corre con el riesgo total de una baja productividad ya que no puede trasladar al precio del proyecto un desvío en horas hombre. En cambio, si el modelo de contratación es por horas hombre (lo que comúnmente se denomina **Time & Materials**) el riesgo es compartido, aunque un aumento de horas hombre por baja productividad será aceptado por el cliente hasta cierto punto y luego habrá problemas de todas formas.

En cualquiera de los casos, un problema de productividad tendrá un impacto en las variables del negocio, que podrá ser directo al reducir la rentabilidad del proyecto, o indirecto al complicar las relaciones con el cliente y la concreción futuros negocios y proyectos.

Un problema de productividad, puede traer también algunas complicaciones al negocio que podrán hacerse visibles a futuro, cuando el proyecto ya está terminado y el sistema se encuentra en la etapa de uso productivo. Es interesante explicar en este punto el concepto de la **deuda técnica** como una obligación contraída a futuro, declarada u oculta, consciente

o inconsciente, por la cual la empresa deberá realizar mejoras al sistema que entregó a un cliente por fallas en su implementación, funcionalidades faltantes o limitaciones que cualquier tipo que en un futuro se pondrán de manifiesto. El concepto de la **deuda técnica** en software es similar y es una metáfora del concepto de cualquier deuda económica: es un pasivo, una obligación contraída que la empresa deberá pagar en algún momento y mientras tanto es muy posible que le genere un costo extra en intereses. El pasivo se materializará con las horas hombre que la empresa deberá insumir para resolver el problema en forma definitiva (pagar la deuda); y los intereses son el costo extra que la empresa deberá asumir para convivir con el problema hasta tanto decida solucionarlo (Ej.: horas de soporte, monitoreo, mantenimientos preventivos o correctivos, etc.).

La idea original de la deuda técnica como una metáfora de la deuda financiera, fue formulada por Ward Cunningham[29] y sobre esta idea Martin Fowler [30]desarrolló una mejora al concepto conocida como el "Cuadrante de la Deuda Técnica". Según él, una deuda técnica asumida en un proyecto puede clasificarse de dos formas:

- Según la prudencia en asumirla puede ser **Prudente** o **Imprudente**.
- Según la consciencia en asumirla puede ser **Advertida** o **Inadvertida**.

La combinación de estas clasificaciones produce cuatro situaciones distintas:

- **Deuda prudente y advertida**: Se produce cuando el equipo es conscientes de que está asumiendo un mal diseño para salir de una situación puntual, como una entrega.

[29] Howard Cunningham: es un conocido desarrollador de software Americano nacido en 1949. Se graduó en ingeniería eléctrica y de computación y tiene una maestría en ciencias de la computación de la Universidad de Purdue. Además del concepto de la Deuda Técnica, Cunningham es conocido por ser el creador de la idea de la colaboración en la Web mediante las Wikis, y sus aportes a la creación de los métodos de programación orientada a objetos y el Extreme Programing (XP). Es además uno de los 17 desarrolladores de software que se reunieron en 2001 dando origen a Agile Alliance y Agile Manifesto.

[30] Martin Fowler: nacido en Inglaterra en 1963, graduado en la University College de Londres en 1986, se mudó a Estados Unidos en 1994. Es un conocido desarrollador de software, famoso por sus publicaciones y aportes al mundo de las metodologías ágiles. Junto con Ward Cunningham fue parte del grupo de 17 desarrolladores que crearon la Agile Alliance y Agile Manifesto en 2001.

Posteriormente se debe evaluar la conveniencia de "pagar la deuda técnica" solucionando este problema.

- **Deuda prudente e inadvertida**: Se produce cuando, con el paso del tiempo, se descubren cómo se deberían haber hecho las cosas. Inevitable, ya que según avanza el proyecto, se aprende de él. Llegado el momento, también se debe evaluar si resulta interesante mejorar el diseño con lo que se ha aprendido o si esto implicaría un esfuerzo demasiado alto.

- **Deuda imprudente y advertida**: Es cuando se deshecha el diseño porque se pretende acelerar el desarrollo. Se deja de hacer diseño, solo se avanza en codificación. Esto se puede hacer hasta un punto determinado del proyecto en el que el beneficio obtenido será mucho menor que los problemas que se generan.

- **Deuda imprudente e inadvertida**: No se conocen las técnicas de diseño ni se es consciente de que se están tomando decisiones incorrectas. Esta deuda es la más peligrosa ya que genera un costo oculto que a futuro habrá que pagar y que no se está teniendo en cuenta.

Conviene siempre saber en qué situación se encuentra el equipo o proyecto respecto de la deuda técnica, y mantenerla controlada (lado consciente del cuadrante). Si bien las tres primeras situaciones se pueden asumir en un momento determinado dentro de un proyecto, la última resulta mucho más peligrosa, porque implica no saber hacia dónde se dirige el proyecto.

Las empresas que presentan problemas de productividad, suelen entregar productos cuyo nivel de **deuda técnica** es alto por aquellas cosas que no pudieron ser implementadas en forma correcta. Esta deuda está allí, es algo latente y basta con que alguna situación o condición de uso del sistema la ponga de manifiesto para que la empresa deba comenzar a pagarla (horas hombre para solucionar el problema). Esta deuda técnica que la empresa debe afrontar aumentará las horas hombre del proyecto original (costo directo del

proyecto), y no permitirá a la empresa dedicar esos recursos al desarrollo de nuevos proyectos (costo de oportunidad). En cualquiera de los casos habrá un impacto negativo en la rentabilidad del proyecto y de la empresa a consecuencia de una baja productividad.

Mantener y mejorar los niveles de productividad permitirá mantener los proyectos dentro de los costos estimados, obtener los niveles de rentabilidad deseados, asegurar el éxito y la continuidad del negocio.

CAPITULO 6. METODOLOGIAS Y PRODUCTIVIDAD

En el mundo del desarrollo de software, y desde los inicios de la actividad, han aparecido un número no menor de técnicas y metodologías que de diferentes maneras han prometido ser la solución a los problemas de productividad que desde siempre se presentaron en esta industria. Pero cómo ya se ha mencionado antes, haciendo referencia a las palabras de Frederick Brooks[31] en su famoso artículo, **No Silver Bullets - Essence and Accidents of Software Engineering**, el tiempo ha demostrado que no hay y posiblemente no haya nunca una sola técnica o metodología, la bala de plata, que genere por sí misma una mejora importante o que solucione los problemas de productividad. Posiblemente sea la suma y la buena combinación de diferentes prácticas y métodos que ataquen la complejidad esencial del software las que puedan generar una mejora importante.

Por esta razón se plantea este capítulo como un trabajo de estudio e identificación de las mejores prácticas que proponen las metodologías más utilizadas, intentando determinar cuáles pueden generar un impacto positivo importante en la productividad. Identificar estas prácticas es el primer paso para pensar luego en la mejor forma de combinarlas en un proceso de desarrollo de software que tenga en cuenta las particularidades del contexto, las características del software a desarrollar, y las habilidades del equipo de trabajo.

Con esta idea de identificar las mejores prácticas de la industria del desarrollo de software, y tratando de identificar aquellas que realmente atacan la complejidad esencial que menciona Brooks, en este capítulo se analizarán diferentes metodologías, técnicas y modelos de procesos para desarrollar software, buscando identificar cómo cada una de ellas promete un aumento o mejora de la productividad si se siguen sus prácticas y lineamientos. Entender cuál es el enfoque mediante el cual cada una de ellas pretende mejorar la productividad, va a ayudar a identificar y listar las buenas prácticas que luego podrán ser

[31] Frederick Phillips Brooks, Jr.: es un ingeniero de software y científico de la computación, nacido en Carolina del Norte (USA) en 1931. Se hizo conocido por dirigir el desarrollo del sistema operativo IBM OS/360 y luego por su libro titulado The Mythical Man-Month (El Mítico Hombre Mes) donde escribe en forma honesta y abierta sobre algunos problemas del desarrollo de software. De este libro sale la famosa Ley de Brooks que dice que "Agregar gente a un proyecto atrasado solo lo atrasará más". También fue famoso y controvertido su paper titulado No Silver Bullet que se menciona en este trabajo.

seleccionar y combinadas en un único proceso adaptado a las necesidades y particularidades de una empresa.

Para estudiarlas, primero serán identificadas y a agrupadas según su estilo, buscando incluir en esta lista las metodologías más difundidas hoy en la industria del software. Esta lista surge de la experiencia profesional del autor, de la literatura analizada, y de las entrevistas y respuestas obtenidas a la encuesta realizada a otros profesionales de la industria.

Según lo mencionado en el párrafo anterior, la lista queda conformada de la siguiente manera:

- Las Tradicionales: Waterfall, RUP, CMMi.
- Las Ágiles: Scrum, eXtreme Programming, TDD, Lean Software Development.

No es el objetivo de este capítulo hacer un análisis o explicación profunda de cada metodología, ni de sus técnicas o herramientas, sino simplemente buscar, identificar y describir los aspectos principales en los que cada una hace foco para lograr buenos niveles de productividad.

Las Tradicionales

Son las metodologías de desarrollo de software que llevan más años en la industria del software y que en general apuntalan la idea de proyectos con planificación a largo plazo, estabilidad de los requerimientos, etapas diferenciadas, roles y responsabilidades formalmente asignados, procesos y documentos definidos entre otras cosas.

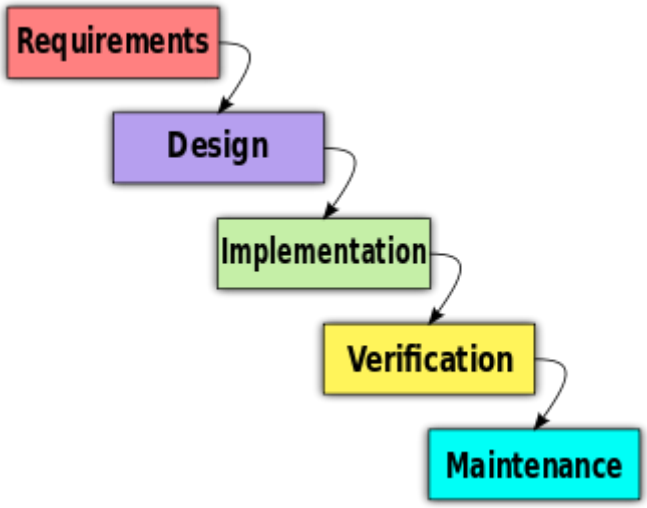
Waterfall

El modelo Waterfall es uno de los más viejos que existen en esta actividad y también es importante mencionar que ha sido muy criticado y está cayendo en desuso. Igualmente es importante analizar algunos de sus aspectos que hacen a la productividad ya que hay cuestiones que siguen siendo hoy la base de algunas prácticas que todavía se utilizan; y otras que han dado origen a prácticas completamente opuestas en las metodologías más modernas. Waterfall es un enfoque lineal y secuencial para desarrollar software, cuya idea original fue tomada de la industria de manufactura, en la cual cada una de las etapas y actividades es realizada una sola vez, y solo cuando la actividad anterior ha finalizado y pasado por una serie de controles y revisiones de calidad. Este modelo de desarrollo (en su implementación más pura) desaconseja fuertemente el regreso a una actividad una vez que esta fue finalizada y esta inflexibilidad fue siempre su aspecto más criticado y lo que hace que hoy este modelo no sea muy aplicable. Por ejemplo, no es posible comenzar con la codificación hasta tanto no se haya finalizado el diseño del software, y no debería ajustarse el diseño una vez iniciada la codificación.

Cada actividad o etapa que finaliza debe pasar por revisiones, verificaciones y validaciones antes de poder iniciar la etapa que sigue, y esto fundamentalmente ayuda a disminuir el riesgo de avanzar sobre resultados de etapas anteriores que puedan contener errores. Esta forma de trabajo claramente ayuda a disminuir riesgos, pero los tiempos de entrega resultantes suelen estar desalineados con las necesidades de los proyectos y las expectativas de los clientes. Se debe dedicar un tiempo y esfuerzo no menores a las verificaciones de cierre de cada una de las etapas ya que la información que en esta se genere será la final y definitiva para el resto del proceso de desarrollo.

Las etapas típicas de un proceso Waterfall son la planificación del proyecto, la definición de los requerimientos, el diseño del software, la implementación o codificación , el testing, la instalación y el mantenimiento.

Figura 7: Waterfall y sus etapas



Fuente: J.D. Mason, *Waterfall to Lean*, www.ceptara.com, 2013.

El principal argumento que dan los que defienden esta metodología de desarrollo es que la inversión de tiempo en las etapas iniciales del proyecto luego permite avanzar con seguridad y minimiza el re-trabajo en etapas posteriores producto de errores conceptuales que no fueron detectados a tiempo. Steve McConnell[32], en su libro "Rapid Development: Taming Wild Software Schedules", estimó que "*...un error de requerimientos que no es detectado hasta las etapas de construcción o mantenimiento, costará entre 50 y 200 veces más para ser solucionado que si hubiera sido solucionado en la misma etapa de requerimientos*".

Esto coincide con la visión de Frederick Brooks, el autor del paper No Silver Bullets, que menciona que "*la parte dura de hacer software es la especificación, diseño y testing de la construcción conceptual, no el trabajo de representarlo y testear la fidelidad de la representación. Todavía cometemos errores de sintaxis, seguramente, pero estos no son nada comparados con los errores conceptuales en la mayoría de los sistemas*".

[32] Steve McConnell: conocido por ser el autor de muchos libros sobre ingeniería de software, incluyendo Code Complete, Rapid Development y Software Estimation. En 1998, Steve McConnell fue nombrado como una de las tres personas más influyentes en la industria del software por la revista Software Development Magazine, junto a conocidos personajes como Bill Gates y Linus Torvalds. Actualmente se desempeña como CEO y Chief Software Engineer en la empresa Construx Software.

Otro argumento de los defensores del modelo Waterfall es que este modelo pone foco en la documentación como base para asegurar la correcta realización de cada etapa y la posibilidad de poder realizar controles de calidad y refinamientos a la información generada que luego será usada por las etapas posteriores. Contar con buena documentación de los requerimientos y del diseño del sistema ayuda a los programadores a realizar una implementación correcta, además de minimizar los atrasos y problemas que pudiera producir la rotación del personal.

RUP – Rational Unified Process

De las metodologías tradicionales de desarrollo de software, RUP es la más utilizada en los últimos años para el desarrollo orientado a objetos. RUP o Rational Unified Process es una metodología desarrollada por la empresa Rational Software que hoy es propiedad de IBM. RUP es un proceso disciplinado que enfoca el desarrollo de software a la asignación formal de roles y tareas al equipo y a la división del trabajo en etapas y fases bien diferenciadas y planificadas (presupuestos y tiempos predecibles) sobre las que se recomienda realizar varias iteraciones. Se trata de un proceso incremental que lleva a una construcción del producto por pasos a medida que se van madurando las definiciones y modelos.

Según la filosofía de RUP[33], la forma de mejorar la productividad del equipo es proveyendo a cada miembro con acceso fácil al conocimiento necesario sobre el sistema a desarrollar mediante documentación de todo tipo: guías, templates, modelos, gráficos, casos de uso y un completo set de información desarrollada según el standard UML. Esto permitiría que todos los miembros de un equipo compartan un lenguaje común y la misma visión y entendimiento del software, además de compartir la forma de trabajar ya que todos seguirán el mismo proceso.

[33] RUP está claramente descripto en *RUP-Best Practices for Software Development Teams*, Rational Software White Paper, USA, 2001.

RUP se basa y hace foco en 6 mejores prácticas para lograr una buena productividad del equipo de desarrollo:

Desarrollar el software en forma iterativa

La complejidad del software de hoy no permite de ninguna forma realizar en forma completa el entendimiento del sistema, para luego modelarlo en forma completa, para luego construirlo, testearlo y entregarlo en forma completa una sola vez. El enfoque iterativo de RUP lleva a lograr un sucesivo refinamiento del entendimiento y de la implementación a través de un desarrollo cíclico donde el cliente debe estar involucrado validando lo que se hace y aportando más información.

Administrar los requerimientos

RUP define como se deben elicitar (redactar), documentar y administrar las definiciones funcionales del sistema a desarrollar usando requerimientos, casos de uso y otras entidades definidas en el modelo UML. El principal beneficio buscado es llegar a implementar un sistema que cumpla con las necesidades de los usuarios de la mejor forma posible habiendo definido y utilizado los requisitos durante todo el ciclo de desarrollo.

Usar arquitecturas basadas en componentes

RUP pone foco en el desarrollo de la arquitectura como una de las primeras actividades de un proyecto entendiendo que es una forma de lograr un sistema modular, fácil de entender y evolucionar, y donde se haga el mayor aprovechamiento posible del reúso de componentes para reducir los tiempos de desarrollo y aumentar la productividad.

Modelar el software en forma visual

RUP pone foco en el modelado gráfico (mediante UML) para definir, validar y comunicar la composición del sistema y su comportamiento, cómo sus componentes calzan y se integran para lograr el funcionamiento esperado. El modelado gráfico con una nomenclatura clara, definida y conocida por todos, reduce la posibilidad de

cometer errores de interpretación e implementación de las funcionalidades. UML se basa para esto en una serie de gráficos estandarizados como son los diagramas de contexto, diagramas entidad relación, diagramas de flujo, diagramas de clases, diagramas de máquinas de estado, diagramas de secuencia, etc.

Verificar la calidad

RUP pone foco en la verificación de la calidad del software como una actividad planificada que es parte del propio proceso y que está presente en todas las actividades y desde el inicio. Es una parte integrante del proceso y es central. No se trata de algo secundario que va en paralelo. De esta forma se logra (o al menos se intenta) realizar la detección y remoción de los defectos en forma temprana. Esto aplica a todos los aspectos de la calidad: funcionalidad, performance, escalabilidad, mantenibilidad, reúso, etc.

Controlar los cambios

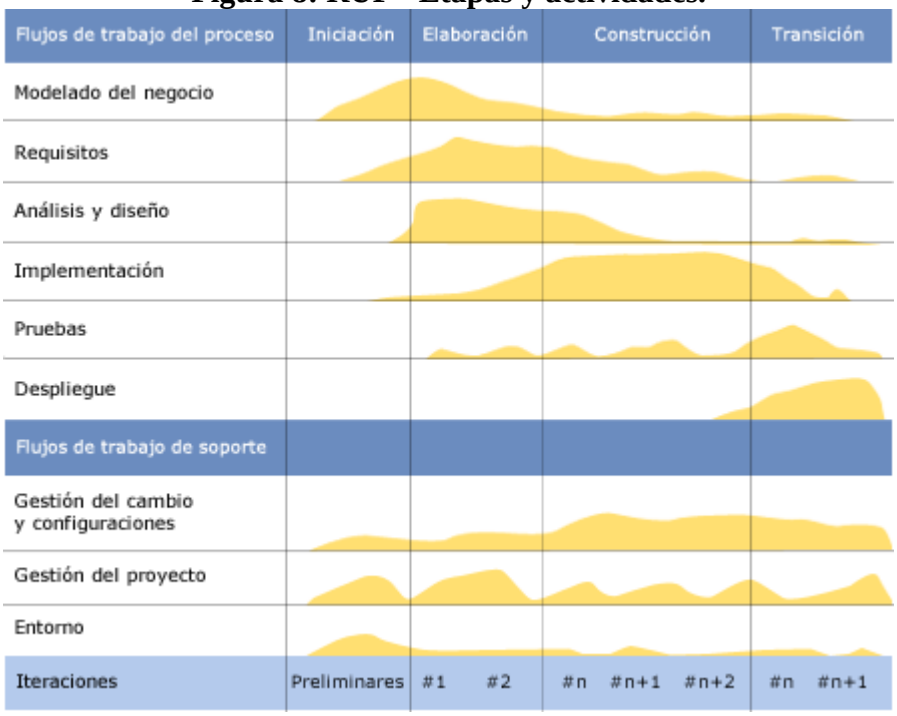
Para RUP los cambios son aceptables ya que son además inevitables; pero deben ser administrados de una manera que permita identificarlos lo antes posible, analizarlos, y autorizarlos para poder ser incorporados al proceso en forma lógica y coherente según la evolución de las iteraciones y entregas planificadas. Los cambios deben ser seguidos para asegurar en todo momento con qué versiones del software y de las especificaciones se está trabajando, y en qué momento y versiones cada cambio será impactado.

Otro aspecto interesante a tener en cuenta de RUP es cómo el proceso se organiza en el tiempo mediante la planificación de las actividades que se agrupan en 4 fases:

- Iniciación
- Elaboración
- Construcción
- Transición

Cada una de las fases como se muestra en el gráfico que sigue, tiene más de algunas de las actividades que de las otras, aunque en general todas las actividades están presentes en todas las fases, justamente por el enfoque iterativo.

Figura 8: RUP - Etapas y actividades.



Fuente: Rational Software, *RUP-Best Practices for Software Dev Teams*, USA, 2001, *p*. 10.

Este enfoque supone una planificación detallada de las fases y las actividades a realizar, quiénes serán los responsables de realizarlas y en qué momento. Este es otro aspecto sobre el que RUP está fuertemente basado para lograr que los tiempos y el presupuesto del proyecto sean predecibles, cosa que se menciona como una de sus grandes ventajas.

CMMi para Desarrollo (CMMi-DEV)

Fue desarrollado por el SEI (Software Engineering Institute) de la Universidad Carnegie Mellon. CMMi-DEV no es un proceso de desarrollo de software ni tampoco es una metodología. CMMi es en realidad un modelo de capacidad y madurez que da a las

empresas una lista ordenada y priorizada de las prácticas y actividades que no pueden faltar en un proceso maduro y capaz de producir software con altos niveles de calidad y productividad. Estas prácticas que el modelo CMMi propone son aquellas que fueron seleccionadas, elaboradas y refinadas por el equipo de expertos que desarrolló el modelo, y además son aquellas que han probado ser prácticas eficaces en la industria del software ya que fueron utilizadas en forma exitosa por una importante cantidad de empresas y proyectos de software.

Este modelo tiene 2 objetivos fundamentales:

- Dar a las empresas una guía para la mejora de sus procesos de software a través de un camino propuesto por los niveles de madurez que indican un orden lógico en el que las prácticas deben ser implementadas.
- Ser un punto de referencia para poder evaluar y comparar el nivel de madurez del proceso de software de una empresa. CMMi no solo define el modelo, sino también un método formal para evaluar la madurez de un proceso llamado SCAMPI (Standard CMMi Appraissal Method for Process Improvement).

Cómo he mencionado antes, el objetivo de CMMi es el de guiar a las organizaciones hacia la madurez y capacidad de los procesos de software para que puedan producir software de calidad, en tiempos y presupuestos predecibles y con buenos niveles de productividad.

El modelo CMMi, cuya versión más actual es la 1.3 publicada en 2010, está compuesto por prácticas que están agrupadas en 22 áreas de proceso. De la lectura y revisión de estas áreas de proceso y prácticas es posible identificar los aspectos en los cuales el modelo CMMi pone foco para lograr buenos niveles de productividad. Cada área de proceso apunta a un aspecto general que es identificado en el objetivo específico del área y luego llevado a detalle en cada práctica. A continuación un resumen de los aspectos que considero los principales respecto del objetivo general de lograr una buena productividad, tomando como base las áreas de proceso de los niveles hasta el 3 inclusive:

- **Project Planning (PP):** establecer una planificación para definir cuáles son las actividades que deben ser realizadas, cuáles serán los productos a entregar, quienes deberán realizar las actividades y cuando.
- **Integrated Project Management (IPM):** lograr una gestión de proyectos que involucre a todos los interesados, que tenga en cuenta los aspectos relevantes de las otras áreas de conocimiento que intervienen en el proyecto, y que el proceso del proyecto sea adaptado a las necesidades puntuales y a las particularidades del mismo.
- **Risk Management (RM):** identificar los posibles problemas antes de que ocurran, identificar y planificar las actividades necesarias como para anular o mitigar el impacto de estos problemas en el proyecto.
- **Requirements Management (REQM):** identificar y gestionar los requerimientos de los usuarios para asegurar que el producto desarrollado cumplirá estos requerimientos.
- **Project Monitoring and Control (PMC):** mantener un adecuado nivel de monitoreo y seguimiento sobre el avance del plan del proyecto y asegurar que se tomen acciones correctivas tempranas si el proyecto no avanza según lo planeado.
- **Configuration Management (CM)**: establecer, mantener y controlar la integridad de los productos del proyecto (finales o intermedios) usando técnicas de identificación y versionado de todos los componentes.
- **Measurement and Analysis (MA)**: desarrollar y mantener capacidades de medición y análisis sobre los aspectos importantes de los proyectos y procesos de software para dar soporte numérico e información objetiva para la mejora de los procesos y la toma de decisiones de gestión.
- **Requirements Development (RD):** profundizar en el conocimiento de qué es lo que el producto debe hacer, mediante una técnica formal de elicitación, análisis y desarrollo de los requerimientos.
- **Validation (VAL):** asegurar que el producto hará lo que tiene que hacer ya que los requerimientos del usuario han sido bien interpretados y documentados. Es muy importante la participación del cliente para asegurar este objetivo. Esta es una de las

formas en las cuales CMMi pone foco para detectar los defectos en forma temprana y reducir el re-trabajo.

- **Technical Solution (TS):** utilizar técnicas formales de ingeniería de software que permitan encontrar las soluciones más adecuadas a los requerimientos. Esto implica utilizar técnicas de arquitectura, análisis y diseño de los componentes del producto, que en su conjunto deberán satisfacer todos los requerimientos.
- **Product Integration (PI):** los componentes que fueron desarrollados por separado deben ser integrados para asegurar de que funcionan juntos como se espera y el producto en su conjunto se comporta según los requerimientos.
- **Verification (VER):** asegurar que el producto desarrollado, cumple con los requerimientos. Esta área de proceso se corresponde mayormente con las actividades de testing en la mayoría de los procesos de desarrollo, pero hace foco también en numerosas actividades ubicadas estratégicamente en diferentes momentos del ciclo de vida del proyecto para favorecer la detección temprana de los defectos y reducir de esta forma el re-trabajo.
- **Organizational Process Definition (OPD):** que la organización defina y mantenga un set de procesos estandarizados para sus proyectos de software, complementados por estándares, guías y políticas útiles para el personal.
- **Organizational Process Focus (OPF):** que la organización ponga foco y atención en el desarrollo, mantenimiento y mejora de sus procesos, basándose en la experiencia de usarlos e identificando sus debilidades y fortalezas.
- **Organizational Training (OT):** poner foco en el desarrollo de las personas, a través de la planificación y realización de actividades de entrenamiento que ayuden a mejorar sus skills y conocimientos para un mejor desempeño de sus funciones.
- **Supplier Agreement Management (SAM)**: asegurar la calidad de los productos o sub-productos desarrollados por terceros a través de la planificación de las entregas, la definición de criterios de aceptación, verificaciones y validaciones de calidad y la gestión de los contratos.
- **Process and Product Quality Assurance (PPQA):** que el personal del proyecto y el equipo de management tengan conocimiento y capacidad de acción respecto del

nivel de calidad de los productos del proyecto y del nivel de apego del mismo a los procesos definidos.

- **Decision Analysis and Resolution (DAR):** que las decisiones importantes en los proyectos se tomen evaluando las alternativas posibles respecto de criterios definidos, usando un proceso de evaluación formal. También es importante que las decisiones queden documentadas para futura referencia y como experiencia útil para futuros proyectos.

Según el reporte titulado **Performance Results of CMMI® Based Process Improvement**[34] que fue publicado por el SEI en 2006, sobre la base de un estudio a 35 empresas, los autores del reporte concluyen en que las empresas que han usado CMMi como modelo base para la mejora de sus procesos han obtenido importantes y visibles mejoras en las 5 categorías de performance que se analizan en el reporte. Las mejoras obtenidas son en promedio las que muestra el siguiente cuadro:

Cuadro 14: CMMi based Process Improvement

Categoría	Porcentaje
Costo del Proyecto	34
Cumplimiento de Fechas	50
Productividad	61
Calidad del Software	48
Satisfacción de los Clientes	14

Fuente: SEI-CMU, *Performance Results of CMMI® Based Process Improvement*, USA, 2006, p. 11.

Las Agiles

Menciono como "las ágiles" a las metodologías que en la industria así son conocidas. Han sido definidas y popularizadas en los últimos 15 años y bregan por su capacidad de

[34] Publicado por el SEI en Agosto de 2006, este informe fue realizado por Diane Gibson; Dennis Goldenson y Keith Kost. El objetivo es reunir y mostrar evidencia sobre los resultados que obtienen las empresas que usan CMMi como modelo de base para la mejora de sus procesos de software. Informe completo en este link.

responder en forma efectiva a los cambios de requerimientos, tomándolos como parte natural del proyecto. Basan sus principios en el **Manifiesto para Desarrollo de Software Ágil** publicado en 2001[35], y que ya se ha mencionado en capítulos anteriores.

Scrum

Scrum es una forma de trabajo ágil para la gestión de proyectos, especialmente diseñada para proyectos complejos. Fue pensada originalmente para proyectos de software, pero la realidad y la experiencia han demostrado que puede utilizarse también para proyectos de otras disciplinas. Scrum se basa en un concepto simple de desarrollo de un producto en incrementos cortos que deben entregar valor al usuario final o cliente. Aquí se presenta el funcionamiento de Scrum en forma muy resumida:

- Un responsable de producto (Product Owner) crea una lista priorizada de funcionalidades a desarrollar que en Scrum se llama Product Backlog.
- En la planificación de cada ciclo de desarrollo (Sprint), el equipo toma algunos puntos de los más prioritarios del Product Backlog y define como los va a desarrollar en el presente Sprint. Con esto se forma el Sprint Backlog.
- El equipo tiene una determinada cantidad de tiempo para completar el trabajo del Sprint, generalmente es una cantidad corta de tiempo, de 2 a 4 semanas. Durante el Sprint el equipo realiza reuniones cortas todos los días, las Daily Meetings o Daily Scrums, para seguir de cerca el avance y detectar rápidamente cualquier inconveniente.

[35] Del 11 al 13 de Febrero del 2001 se reunieron en el Snowbird Ski Resort, en las Montañas Wasatch Utah, un grupo de 17 desarrolladores de software experimentados con la idea de encontrar formas de desarrollar software alternativas a las tradicionales que ellos llamaban "heavyweight". Aquel grupo se autodenominó The Agile Alliance, y lo que surgió de la reunión es el Agile Software Manifesto, un documento público con los 12 principios de lo que a partir de ese momento se llamaría Agile Software Development. Aquel grupo de 17 desarrolladores estuvo formado por: Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland y Dave Thomas.

- Durante el desarrollo del Sprint, el Scrum Master se encarga de mantener al equipo enfocado en el desarrollo, solucionando cualquier problema o interrupción que pueda ocurrir para que el equipo no pierda el foco y que el desarrollo no se demore por ninguna razón.
- Al final del Sprint se debería poder obtener un resultado que pueda ser entregado al cliente o mostrado a los interesados en el proyecto. De alguna manera el resultado del Sprint debe aportar funcionalidad o valor. Este es un aspecto fundamental en Scrum, se debe entregar valor desde el primer Sprint.
- Cada Sprint finaliza con una reunión de revisión de lo hecho, la llamada Retrospectiva, que sirve para analizar lo que se hizo, qué cosas salieron bien y/o mal, y cómo se puede mejorar en los próximos Sprints.
- El equipo vuelve a elegir funcionalidades prioritarias desde el Product Backlog, se planifica el próximo Sprint, y así el proyecto continúa hasta que se consume todo el Product Backlog, o hasta que alguna otra condición indica el fin del proyecto (tiempo, presupuesto, etc.). No importa que condición indique el fin del proyecto, el método asegura que el trabajo más prioritario debería estar realizado.

Figura 9: SCRUM - Esquema del método

Fuente: www.scrumalliance.org

De la lectura y análisis de varios artículos sobre Scrum[36], surgen los siguientes puntos como los más importantes sobre los que el método se basa para lograr buena productividad:

- Scrum define actividades de gestión y seguimiento con muy alta frecuencia (todos los días) para poder hacer un seguimiento del proyecto en tiempo real, detectar los problemas cuando ocurren y solucionarlos rápido, antes que afecten el avance del proyecto.
- En Scrum, la unidad central de organización es el equipo, no el proyecto. Se busca que los equipos sean permanentes y que los proyectos pasen por los equipos en lugar de formar equipos para los proyectos y disolverlos al terminar. Esto permite que se cumpla el ciclo de vida de la formación y consolidación de un equipo de trabajo para llegar a lograr un funcionamiento óptimo.
- Se rompe la separación entre la definición y documentación de los requerimientos y el desarrollo que se hace a partir de estos documentos en los procesos de tipo Waterfall o RUP. En Scrum el equipo completo trabaja junto e integrado compartiendo el conocimiento y refinando el producto en cada Sprint.
- Involucramiento de todo el equipo del proyecto en la definición de los User Stories, las estimaciones, la planificación de los Sprints y la distribución del trabajo. Esta práctica ayuda a aumentar el compromiso ya que el trabajo no es asignado al equipo desde afuera, sino que el propio equipo es el que se asigna el trabajo, estima los tiempos y lo distribuye entre sus miembros.
- Entrega de funcionalidad al cliente/usuario en forma frecuente lo que ayuda a mantener al cliente/usuario involucrado e interesado con el proyecto y permite identificar en forma temprana cualquier desvío respecto de sus deseos, expectativas y necesidades.
- Frecuencia, fluidez y transparencia se fomenta en las comunicaciones en el equipo, materializadas especialmente en las Daily Meetings donde cada miembro del equipo

[36] Artículos publicados en la Revista especializada Agile Journal:
- How Scrum Generates Increased Productivity, Part 1: The Scrum Master. USA, April 2009.
- How Scrum Generates Increased Productivity, Part 2: The Product Owner. USA, May 2009.
- How Scrum Generates Increased Productivity, Part 3: The Team. USA, June 2009.

debe contar rápidamente qué logró hacer ayer, qué tiene planeado hacer hoy, y qué dificultades tiene para avanzar.

- Es muy típico el trabajo en parejas (Pair Programming) lo que ayuda a mejorar la calidad del producto y a lograr una mejor distribución y propagación del conocimiento ya que las parejas de trabajo deben rotar frecuentemente. Pair Programming es una práctica típica de XP (eXtreme Programming), una metodología de desarrollo de software que se acopla muy bien con Scrum.
- El cliente involucrado en el proyecto, en la persona del Product Owner, asegura que siempre está presente la visión del cliente en el proyecto y que el avance se hace firme hacia lo que el cliente realmente quiere y necesita.
- El Scrum Master que juega un papel muy importante manteniendo la comunicación y la coordinación entre el Product Owner y el equipo, y asegura que el equipo se mantenga enfocado en el desarrollo eliminando problemas, obstáculos y distracciones.
- Mediciones y documentación livianas (que no generan sobre-esfuerzo para poder generarlas y mantenerlas) y permiten seguir el avance del desarrollo y ajustar si es necesario en forma rápida (de un Sprint al otro).
- Estabilidad de requerimientos del Sprint, el Product Owner puede trabajar en los requerimientos del próximo Sprint, pero no puede modificar los del Sprint que está en curso. Esta es una regla de oro de Scrum que permite al equipo avanzar sin interrupciones. Los cambios o modificaciones deben ser incluidos en el próximo Sprint, los tiempos cortos permiten que esto pueda hacerse sin grandes esperas o atrasos.

eXtreme Programming (XP)

XP es una metodología de desarrollo de software ágil que para describirla la literatura especializada usa la palabra en inglés "lightweight"[37]. La traducción literal de esta palabra

[37] "*XP is lightweight, In XP you only do what you need to do to create value for the customer*", Extreme Programming Explained: Embrace Change, Kent Beck, Cynthis Andres. USA, 2004, *p* 3.

sería liviana o ligera, lo que indica que se trata de una metodología despojada de todo aquello que podría considerarse innecesario o no imprescindible. XP pone especial énfasis en remarcar que la actividad principal es la codificación y por eso se deja afuera todo lo que no sea codificar, lo que no tenga una relación estrecha con esto, o lo que no pueda hacerse mientras se codifica. La planificación a largo plazo, las extensas actividades de diseño o arquitectura o la documentación detallada son ejemplos de estas actividades que en XP podrían considerarse como "heavyweight" y son tratadas de otra manera.

XP está especialmente pensada para equipos de desarrollo pequeños o medianos, desde 2 hasta 10 desarrolladores, donde la comunicación directa y fluida pueda reemplazar a los extensos documentos y permita afrontar el dinamismo que impone un entorno donde los requerimientos cambian rápida y frecuentemente.

El nombre de eXtreme Programming, proviene de un concepto fundamental de esta metodología que se trata de llevar al extremo la práctica de algunos principios que han demostrado ser de utilidad y que son claves para lograr buenos resultados. Algunos ejemplos para entender mejor la filosofía XP:

- Si las revisiones de código son buenas, entonces en XP se las lleva al extremo practicándolas todo el tiempo. Se revisa el código en forma constante a medida que se genera y a esto se le llama **Pair Programming**.
- Si el testing es bueno, OK, entonces todos los desarrolladores deben testear todo el tiempo sus piezas de código. A esto se le llama **Unit Test**.
- Si el diseño es bueno, entonces en XP se lo practica todo el tiempo como parte de las actividades que todo desarrollador debe realizar a medida que su código crece para mantener la simplicidad. A esto se le llama **Refactoring**.
- Si la arquitectura es importante, entonces todos deben trabajar en definir y refinar la arquitectura todo el tiempo. A esto se lo denomina **Metaphor**.
- Si el testing de integración es importante, entonces por qué esperar al final del proyecto para hacerlo solo una vez. En XP se integra y prueba varias veces al día, con cada nueva versión de un componente. Esto se llama **Continuous Integration**.

- Si las iteraciones cortas que permiten entregas rápidas son buenas, entonces XP hace las iteraciones realmente cortas. Segundos, minutos u horas. Ni semanas ni meses. A esto se le llama **The Planning Game**.

Otra visión interesante para entender la filosofía de XP es preguntarse qué cosas haría nuestro equipo en un proyecto de desarrollo de software si tuviera mucho tiempo. Las clásicas respuestas serían: una buena planificación, una arquitectura detallada, un muy buen diseño, revisiones profundas del código, mucho testing, se pondría foco en la comunicación con el cliente y con el equipo y por último se harían mejoras al sistema a medida que se van aprendiendo cosas nuevas sobre su funcionamiento y sobre el negocio. Lamentablemente el tiempo no suele ser un recurso abundante, y estas mismas cosas son las que normalmente se dejan de hacer o se descuidan a niveles peligrosos para el proyecto. XP se basa justamente en qué; como no hay tiempo suficiente; hay que poner especial foco en estas actividades y hacerlas de manera extrema, todo el tiempo, durante la codificación.

A continuación un poco más de detalle y explicación sobre las prácticas mencionadas antes, que son la base sobre las que XP busca maximizar la productividad:

- **The Planning Game** — Se trata de determinar en forma rápida el alcance del próximo release con la combinación de las funcionalidades a desarrollar que son más prioritarias para el negocio y las estimaciones hechas por el equipo de desarrollo. Planificar iteraciones cortas también permite reaccionar rápidamente a los cambios actualizando el plan.
- **Small Releases** — Se trata de poner en producción una versión simple del sistema en forma rápida con las primeras funcionalidades. Luego entregar nuevas versiones en ciclos cortos completando y mejorando la funcionalidad.
- **Metaphor** — Se trata de guiar el proceso de desarrollo con una explicación simple y compartida por todos de cómo tiene que trabajar el sistema. Es como contar un cuento, manteniendo la simplicidad y la claridad.
- **Simple Design** — Se trata de mantener en todo momento el diseño más simple posible, suficiente para que el sistema funcione y haga lo que tiene que hacer sin

complejidad extra. Cualquier complejidad innecesaria debe ser eliminada en cuanto es descubierta.

- **Testing** — Se trata de que los programadores debe escribir todo el tiempo los casos de test de unidad que deben ser ejecutados en forma exitosa para poder avanzar. Este testing debe ser re-ejecutado con cada cambio. El cliente también debe realizar test a nivel de funcionalidades para demostrar que estas hacen lo que tienen que hacer y están finalizadas.
- **Refactoring** — Se trata de que los programadores deben reestructurar el código a medida que este crece y se vuelve más complejo, sin alterar su comportamiento. Se busca con esto eliminar complejidad adicional, posibles duplicaciones, mejorar la performance, simplificar y agregar flexibilidad para futuros cambios.
- **Pair Programming** — Se trata de que todo el código debe ser escrito por 2 programadores trabajando juntos sobre una sola computadora. Ambos trabajan juntos pero cumpliendo diferentes roles que aseguran calidad del código y foco en la funcionalidad.
- **Collective Ownership** — Se trata de maximizar el trabajo en equipo, la colaboración y la responsabilidad colectiva sobre el resultado final. Todos pueden (y deben) cambiar el código de cualquier parte del sistema, en cualquier momento, cuando se trata de mejorar algo, solucionar una falla detectada o avanzar en la implementación.
- **Continuous Integration** — Se trata de compilar, integrar, construir y probar una versión del sistema varias veces por día, preferentemente en forma automática, cada vez que algo se cambia o que una tarea de codificación se finaliza.
- **40 Hour a Week** — Se trata de no trabajar más de 40 horas a la semana, es una regla de XP que permite mantener al equipo descansado y altamente productivo. Si hubiera que hacerlo de todos modos, no se debe trabajar tiempo extra 2 semanas seguidas.
- **On-Site Customer** — Se trata de que el cliente, o el usuario final del sistema sea parte integrante del equipo para tener siempre su visión presente y no desviarse de ella. Debe estar disponible full time para responder preguntas y despejar dudas del equipo de desarrollo.

- **Coding Standards** — Se trata de tener ciertas reglas de codificación que faciliten la lectura y comprensión del código por parte de otros. Se trata de comunicarse a través del código para permitir que los programadores puedan entender y continuar el código de otros. Esto suele lograrse, entre otras cosas, con abundantes comentarios en el código.

Se encuentra en estas prácticas una gran similitud con la propuesta de Scrum, y es por eso que ambas metodologías suelen aplicarse juntas en los proyectos de desarrollo de software denominados ágiles.

TDD (Test Driven Development)

Aunque muy asociado a la práctica de Unit Test que forma parte del método XP (eXtreme Programming), TDD no es lo mismo y podría definirse como una forma de desarrollar software independiente de XP que busca mejorar la calidad del código, reducir los errores y en definitiva mejorar la productividad. Aunque su nombre puede hacer pensar que se trata de una forma de testear software, se trata en realidad de una forma de diseñar y desarrollar.

TDD se trata básicamente de desarrollar primero el código necesario para el o los casos de testing unitario que el programa debe pasar en forma exitosa para cumplir con los requerimientos. Por supuesto en su primera corrida estos casos de testing deben fallar porque no existe todavía el código del programa que debe superar esos test. Se conoce esta metodología también como Test First Development, justamente porque se deben primero codificar los casos de test y luego el código del programa. Luego de codificados los casos de testing, el programador debe enfocarse en construir el código necesario para pasar los casos de testing en forma exitosa. El programador debe continuar escribiendo código solo si hay casos de testing que fallan; cuando todos los casos pasan, entonces el programador debe dedicarse a mejorar el código escrito, pero cuidando de no modificar su comportamiento.

En el Libro JUnit in Action[38], sus autores Tahchiev, Leme, Massol y Gregory, explican que: "*Test-driven development (TDD) is a programming practice that instructs developers to write new code only if an automated test has failed, and to eliminate duplication. The goal of TDD is clean code that works*".

Esta característica es la que principalmente diferencia TDD del Unit Test de XP, donde los casos de testing podrían desarrollarse después del código principal, todos juntos al final, incluso podrían ser codificados por otro programador y ejecutados por otra persona.

Según los promotores de TDD, escribir primero el código de los casos de test lleva al programador a definir primero qué es lo que el programa debe hacer, lo que esencialmente es una actividad de análisis. Terminado esto, el programador sabe exactamente y entiende qué es lo que el programa tiene que hacer y avanza en la implementación con los conceptos de funcionalidad clarificados.

TDD es una práctica que recibió mucha atención en el mundo del software como parte fundamental de eXtreme Programming, e incluso se han creado herramientas de desarrollo que facilitan esta práctica, soportando el desarrollo y la ejecución automática de los casos de test con generación de reportes y estadísticas inclusive.

TDD apunta fuertemente a la generación de código sin fallas por el hecho de que el mismo programador va testeando su código a medida que lo genera, pasando cada uno de los casos de test unitario definidos. La codificación y el testing se hacen juntas y terminan al mismo tiempo cuando todos los test pasan y el código está refactorizado.

TDD es también una forma de incorporar actividades de diseño a la codificación ya que a medida que el código crece para poder superar los test, el programador debe mejorarlo

[38] JUnit in Action: un libro que explica el uso de JUnit, un potente framework de test unitario para Java. Se explican también los conceptos fundamentales de la actividad como Unit Test y Test Driven Design. Sus autores: Petar Tahchiev: un ingeniero de software de HP y desarrollador líder del proyecto Jakarta Cactus. Felipe Leme: miembro de la JCP (Java Community Process) y participante de los proyectos DbUnit and Cactus. Gary Gregory: un desarrollador Java con más de 20 años de experiencia. Vincent Massol: el autor de la primera edición del libro Junit in Action.

haciéndolo más claro, ordenado y mantenible, eliminando duplicaciones, agrupando o encapsulando funciones, etc. Esto es conocido como Refactoring y apunta a mejorar la productividad a futuro dejando código más simple y limpio que será fácil de entender, mantener y mejorar.

Las ventajas que TDD aporta en Calidad y Productividad pueden resumirse en estos conceptos que son la base del método:

- **Design & Refactoring**: permite generar código simple y mantenible.
- **Early Testing**: posibilita la detección de los defectos poco después de generarlos.
- **Automated Testing**: permite repetir el testing todas las veces que sea necesario con muy poco esfuerzo.

Just in Time (JIT) - Lean Software Development

Lean o Just in Time son términos con los que la industria nombra a una filosofía de trabajo que nació en el marco de la industria automotriz japonesa de los años 50, con la que se referían a procesos de producción extremadamente eficientes que buscan minimizar o eliminar el desperdicio y cualquier pérdida de tiempo (que también se la considera un desperdicio).

Se considera a Toyota como la creadora de esta corriente de pensamiento en cuanto a sistemas de producción e ingeniería de procesos, ya que fueron los primeros que organizaron su sistema de producción con el objetivo de eliminar todo desperdicio, distinguiendo 7 tipos de desperdicio: defectos, exceso de producción, transporte de materiales de un lado a otro, esperas en la línea de producción, grandes inventarios (stock), movimientos y procesos o actividades innecesarias.

El sistema de producción Toyota es un ejemplo clásico de la filosofía Kaizen (o mejora continua) de mejora de la productividad. Muchos de sus métodos han sido copiados por otras empresas de la misma y de otras industrias, y ahora a este sistema se lo conoce también como Lean Manufacturing (Fabricación Esbelta), ya que refiere a procesos flacos, descargados de toda actividad innecesaria, pérdidas de tiempo, burocracias, etc.

Estos términos Lean y JIT no fueron creados por los Japoneses para denominar sus procesos, sino que fueron popularizados por James Womack, Daniel Jones y Daniel Roos, en el libro que escribieron en el año 1990 titulado "The Machine that Changed the World"[39] en el cual hicieron un profundo análisis de esta forma de producir con la cual Toyota logró niveles de calidad y productividad muy superiores a los de la industria automotriz global de aquellos años. El libro fue escrito como parte de un estudio global de la competencia industrial conducido por el MIT, y se convirtió rápidamente en un Best-Seller de su categoría popularizando estos términos.

La realidad es que Toyota logró con esta forma de trabajo acortar mucho los tiempos de ingeniería y producción de automóviles, al mismo tiempo que pudieron aumentar los niveles de calidad, logrando una ventaja competitiva muy importante respecto de sus competidores, las industrias automotrices americanas y europeas.

Con el tiempo, esta filosofía de producción esbelta llegó a la industria del software, popularizándose como Lean Software Development, o Just in Time Software Development, con fuertes coincidencias con las metodologías ágiles como SCRUM o XP por la similitud de sus principios.

[39] The Machine that Changed the World, un libro fundamental en la historia de los métodos Lean, cuenta la historia del cambio de Toyota, cómo mejoraron su sistema de producción y sus normas culturales. Resalta la manera muy cercana en que Toyota trabajó con sus proveedores para lograr sus objetivos.

Fueron Mary Poppendieck y Tom Poppendieck los que popularizaron el uso de esta filosofía en el desarrollo de software con el libro titulado "**Lean Software Development: An Agile Toolkit**" que ambos publicaron en 2003[40], en el cual muestran cómo aplicar los principios LEAN al desarrollo de software. En este libro los autores definen en una primera aproximación los que llaman los 7 principios del Lean Software Development:

- Optimizar el todo.
- Eliminar desperdicios.
- Calidad en la construcción.
- Aprender constantemente.
- Entregar rápido.
- Involucrar a todo el mundo.
- Seguir mejorando.

Es importante destacar que Lean Software Development no es una metodología en si misma ni tampoco un proceso definido para desarrollo de software. Se trata en realidad de una filosofía sintetizada en una serie de principios, que pueden ser aplicados a las actividades de desarrollo de software para mejorar la productividad y la calidad. Si es importante reconocer que los principios Lean guardan una importante similitud con los principios que definen las metodologías ágiles como Scrum y XP, donde la aplicación de principios Lean se hace más natural por las coincidencias que tienen.

En sucesivas publicaciones de Mary Poppendieck y Tom Poppendieck, los principios Lean fueron variando ligeramente hasta llegar a la forma en que se los conoce hoy que es la siguiente:

[40] Mary y Tom Poppendieck, ambos marido y mujer, experimentados profesionales de la industria del software, con títulos de grado y posgrado en Matemáticas (Mary) y en Física (Tom), vieron la oportunidad de desarrollar un nuevo enfoque para el desarrollo de software, basándose en los principios Lean de la industria manufacturera. Publicaron con esta idea en 2003 el libro de referencia, luego siguieron otros 3 libros: Implementing Lean Software Development: From Concept to Cash (2006), Leading Lean Software Development: Results are Not the Point (2009), y The Lean Mindset: Ask the Right Questions (2013).

Lean Principle #1 – Eliminate Waste

Se trata de identificar y eliminar cualquier trabajo, producto, actividad, o tiempo muerto que genere una demora o un sobrecosto y que no aporte valor al producto o al cliente final. Los 7 tipos de desperdicio identificados por Toyota en su sistema de producción encuentran su similitud en la industria del software en cosas como estas:

- Código o funcionalidades innecesarias.
- Iniciar más actividades de las que pueden ser completadas.
- Demoras, esperas, tiempos muertos.
- Requerimientos que no están claros o cambian frecuentemente.
- Burocracia, excesiva documentación.
- Comunicación lenta o poco efectiva.
- Trabajo que queda sin terminar.
- Errores del software, defectos, bugs.
- Cambios frecuentes entre tareas (task switching).

Las reuniones de retrospectiva que se practican en las metodologías ágiles o las sesiones de lecciones aprendidas de los procesos tradicionales son buenos momentos para poner foco en identificar estos desperdicios y definir la forma de eliminarlos.

Lean Principle #2 – Build Quality In

Los defectos en el software son un claro desperdicio, más aún cuando pasan de una etapa a otra del proyecto y obligan a volver hacia atrás para solucionarlos. Este principio apunta a poner foco en detectar y eliminar los defectos en el momento en el que se generan mediante actividades que permitan que los controles y revisiones se hagan casi en paralelo al desarrollo, al mismo tiempo. Un buen ejemplo de esto pueden ser las actividades de testing unitario y programación por pares que se definen en eXtreme Programming.

Lean Principle #3 – Create Knowledge

Siendo que el desarrollo de software es una actividad basada en el conocimiento y que solo puede ser realizada con él, este principio apunta no solo a generarlo sino también a distribuirlo para evitar que cierto conocimiento quede en una sola persona y que luego esta pueda convertirse en un cuello de botella para nuestro proyecto. Nada se compara al conocimiento que cada desarrollador genera para sí mismo sobre el código que él mismo escribe, pero es muy importante que ese conocimiento llegue a los demás miembros del equipo. Actividades como Pair Programming, revisiones de código, reuniones frecuentes, presentaciones, comentarios en el código o la conformación de una Wiki pueden ayudar mucho en este aspecto.

Lean Principle #4 – Defer Commitment

Decidir lo más tarde posible, de eso se trata. Sin llegar a generar demoras y sin tomar decisiones tardías, este principio indica que se debe decidir lo más tarde posible, especialmente para aquellas decisiones que no tienen retorno o cuyo retorno podría generar un fuerte impacto en el proyecto. Decidir lo más tarde posible permite contar con mayor información y conocimiento sobre el problema y las opciones disponibles. Hay aspectos e información importante que aparecen tarde en los proyectos. Incluso es posible contar con nuevas opciones que no estaban disponibles tiempo atrás. Otro beneficio importante es que se reduce la posibilidad de cambios cuando se decide en forma temprana algo que se va a hacer mucho más adelante. Es mejor esperar todo lo posible y tomar la decisión con más y mejor información y opciones.

Lean Principle #5 – Deliver Fast

En muy pocos casos se tiene el conocimiento exacto de lo que se necesita al principio del proyecto cuando se elaboran los requerimientos. El producto final suele ser el resultado de un proceso donde los requerimientos van madurando y el cliente descubre de a poco lo que quiere y necesita cuando avanza en el conocimiento del problema. Es por eso que desarrollar un producto completo en una sola entrega monolítica lleva a la situación casi segura de tener que hacer cambios. El cliente terminó de entender el problema que tenía y cómo solucionarlo luego de ver algunos

intentos de solución. Por eso Lean recomienda mediante este principio que se hagan ciclos cortos y entregas frecuentes en las que el cliente (y el mismo equipo de desarrollo) pueden ir de a poco ganando conocimiento sobre el problema, madurando en sus ideas e imaginando la mejor solución posible.

Lean Principle #6 – Respect People

Respetar a las personas parece ser una obviedad, algo que ya no es necesario mencionar y que debe darse de manera natural en todo entorno de trabajo, no solo en actividades de software. Pero este principio de Lean va más allá y apunta a otra forma de respeto que es darles a las personas la oportunidad de usar su conocimiento, poner en juego su experiencia y tomar sus propias decisiones acerca de su trabajo, valorando su capacidad y su profesionalismo. Es importante permitir a las personas exponer sus puntos de vista respecto de cómo enfocar la solución a los problemas, abrir espacios de discusión, de intercambio de opiniones y lograr definiciones conjuntas en los aspectos que hacen al software a desarrollar, su arquitectura, diseño, etc.

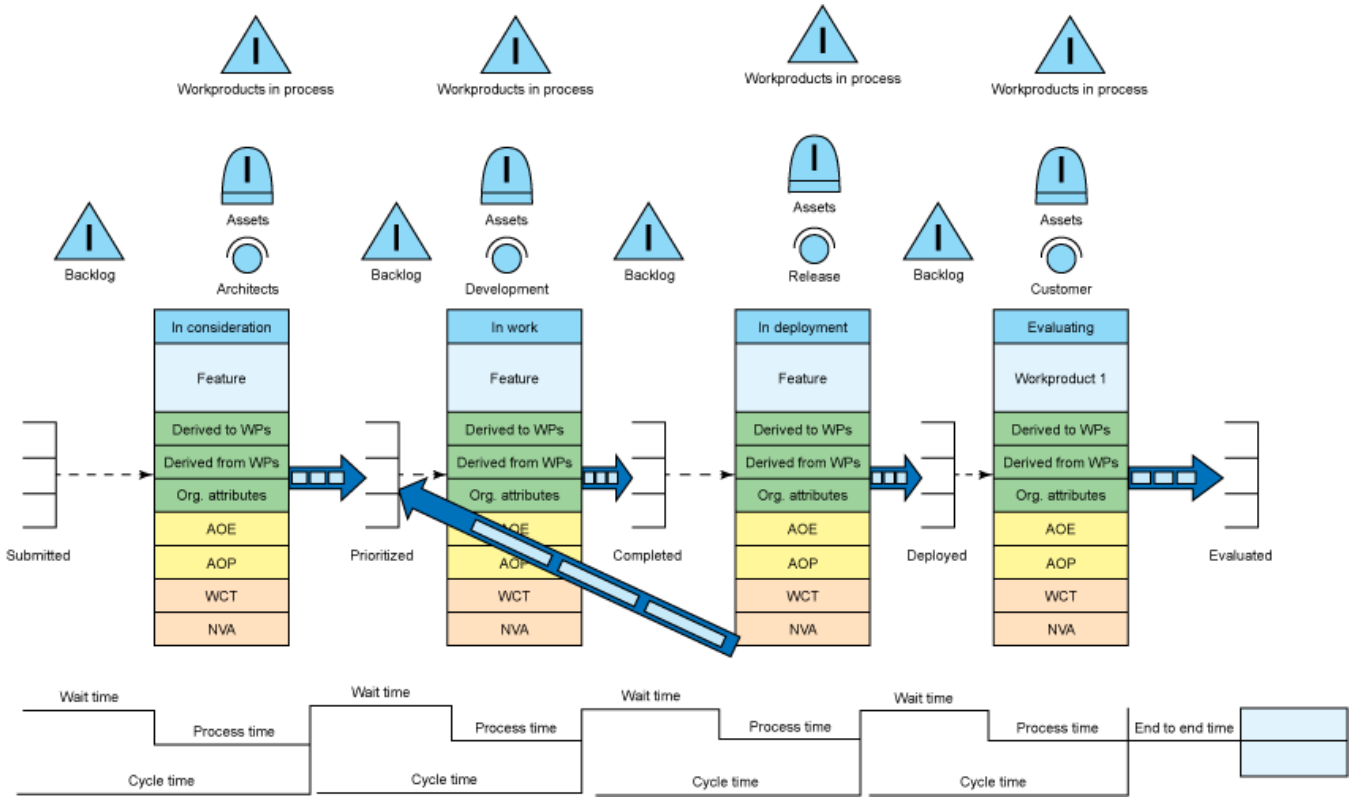
Lean Principle #7 – Optimise The Whole

Es importante poner foco en la optimización general de la cadena de valor en la empresa, y no solo en algunos procesos o equipos en forma individual. La mirada debe ser general y de alto nivel, desde los proveedores y hasta el cliente, pasando por la propia empresa. Es muy común que grandes problemas o demoras sean producto de problemas de coordinación y comunicación entre grupos, departamentos o incluso entre empresas enteras. La mirada de alto nivel permitirá detectar y eliminar los grandes desperdicios que causan el mayor impacto.

Para terminar, Lean se apoya en una serie de herramientas que permiten entender los procesos en una organización, detectar y eliminar los desperdicios. Una de ellas es el Value Stream Mapping, o el Mapa del Flujo de Valor. Se trata de una herramienta gráfica, del tipo diagrama de flujo, que permite analizar el estado actual de un proceso de producción o

desarrollo, desde el proveedor hasta el cliente, con el objetivo de detectar desperdicios y diseñar el futuro proceso en el cual se los elimina.

Figura 10: Lean - Mapa de Flujo de Valor



Fuente: internet, créditos a quién corresponda.

Como cierre de este capítulo, se presenta el siguiente cuadro que muestra las diferencias de enfoque entre las metodologías ágiles y las tradicionales en 14 aspectos fundamentales de la actividad de desarrollo de software. Estas diferencias serán la base del análisis en próximos capítulos:

Cuadro 15: Métodos Tradicionales y Agiles – Comparación

	Las Tradicionales (WF, RUP, CMMi)	Las Agiles (Scrum, XP, TDD, Lean)
Approach General	Actividades en Cascada con una sola entrega al final (WF) o Iterativo con varias entregas.	Puramente iterativo, ciclos muy cortos, muchas entregas.
Planificación y Seguimiento	Planificación completa y detallada, a largo plazo y con fechas de entrega comprometidas e hitos intermedios del control. Reuniones e informes de avance de los proyectos a intervalos regulares.	Solo se planifica el ciclo que va a comenzar, recién al finalizar se planifica el que sigue. Las decisiones se toman lo más tarde posible para contar con más información. Siempre se busca hacer lo más prioritario. Comunicación abierta y fluida, reuniones todos los días para detectar y solucionar problemas.
Gestión del Alcance del Proyecto	Alcance totalmente definido y documentado con detalle, acordado con el cliente, generalmente antes de comenzar. Se controlan las variaciones de alcance con un procedimiento formal y autorizaciones.	El alcance es abierto, se mantienen las funcionalidades en un backlog que puede cambiar todo el tiempo. Solo se busca mantener sin cambios las funcionalidades del ciclo actual.
Gestión del equipo y las personas	Equipo formado para el proyecto, con roles y responsabilidades formalmente asignados y definidos. Cada uno tiene su función que a su vez está bien documentada. Las tareas también se asignan según la planificación y rol de cada uno.	Se busca que el equipo sea estable y los proyectos pasen por él. Equipos pequeños, multidisciplinarios, auto-contenidos y auto-gestionados. Las mismas personas se asignan las tareas según las necesidades. Foco en dar a las personas la posibilidad de hacer uso de sus capacidades y experiencia y en tomar sus propias decisiones.
Roles y Responsabilidades	Formalmente asignadas y descriptas en documentos formales de la organización.	Auto-asignadas por el propio equipo según las necesidades del momento y la experiencia de cada miembro del equipo.
Gestión del Conocimiento	Transferido a través de documentación formal y actividades planificadas.	Se busca que fluya en forma natural, maximizando la comunicación entre las personas, el trabajo en equipo y documentando solo lo estrictamente necesario.
Definiciones Previas a la construcción	Mucho énfasis en invertir tiempo en actividades previas de definiciones y especificación del sistema (requerimientos, arquitectura, diseño, etc.) antes de comenzar la construcción en cada incremento.	Se comienza directamente con la construcción y se va logrando el entendimiento necesario a medida que se avanza. El cliente debe estar involucrado todo el tiempo, refinando su visión de lo que necesita a medida que el proyecto avanza.

Productos Generados (Software y otros)	Además del software, se genera abundante documentación de todo tipo siguiendo estándares y templates. Toda la definición del sistema y el conocimiento obtenido debe quedar documentada para poder ser transferido a otras personas.	Prioridad absoluta a generar el software, y la menor cantidad de documentación posible. El conocimiento debe transferirse a través de trabajo en equipo y colaboración en lugar de documentos. El código debe "hablar por sí mismo" siendo claro y con abundantes comentarios.
Entrega al cliente	Planificadas de antemano, una o varias según el enfoque del proyecto.	Al final de cada ciclo, muchas y muy frecuentes para mantener al cliente involucrado en el proyecto y poder validar que se avanza según su visión.
Gestión de la Calidad	Foco en la detección temprana de los defectos a través de actividades de revisión y pruebas ubicadas en puntos estratégicos del ciclo de vida del proyecto.	Se busca la detección de los defectos, al instante de que estos se generan, con actividades que se realizan junto con la construcción y en forma constante: programación por pares, testing unitario, test driven development, etc.
Gestión de los Cambios	Una vez definido el alcance del proyecto, los cambios se gestionan mediante un proceso formal que asegura que cada cambio sea analizado, presupuestado y autorizado. Cada cambio luego se inserta en el plan del proyecto para que sea realizado con el menos impacto posible.	El cambio se lo considera parte del proceso y se lo trata con naturalidad. El backlog puede cambiar todo el tiempo, las funcionalidades pueden ser reordenadas según cambios de prioridad. Solo lo que está siendo desarrollado en el ciclo actual se lo intenta mantener congelado.
Involucramiento del Cliente	Se lo involucra fuertemente en la etapa de las definiciones para obtener la información necesaria sobre sus necesidades. Luego, mientras el equipo trabaja en la construcción, el cliente no tiene una intervención programada.	Debe estar involucrado todo el tiempo, en contacto con el equipo de proyecto, afinando su visión y aportando información. Recibe entregas frecuentes que debe probar y dar feedback y guiar al equipo de proyecto.
Mejora de los procesos	El proceso standard de la organización debe ser mejorado con la experiencia obtenida en los proyectos.	Al fin de cada ciclo el equipo analiza el resultado obtenido y define que se debe mejorar para el próximo ciclo.
Cumplimiento de los Procesos	Se contempla la realización de controles y auditorías para asegurar que las personas cumplan el proceso y dar visibilidad de esto a la organización.	No se contemplan controles ni auditorías.

Fuente: elaboración propia.

CAPITULO 7. UNIENDO LOS MUNDOS EN UN NUEVO ENFOQUE

Haber desarrollado las explicaciones del Capítulo 6 sobre algunas de las metodologías de desarrollo que hoy se usan, llevó por una extensa recorrida sobre libros, notas y papers que permitió ver en qué aspectos de la actividad cada una pone foco para lograr buenos niveles de productividad. Pero también permitió vislumbrar los 2 mundos del desarrollo de software, mundos en los cuales se pretende llegar a resultados similares con técnicas y métodos que difieren bastante.

De un lado parece que el secreto está en lograr un profundo entendimiento del problema a resolver, junto con una planificación detallada y una adecuada asignación de roles y responsabilidades. Se debería poder de esta forma alcanzar el resultado esperado.

Del otro lado, por el contrario, parece no haber posibilidad de largas planificaciones ni tampoco de entender lo que hay que hacer porque ni el mismo cliente lo tiene claro al principio del proyecto, por lo tanto el secreto está en la habilidad para garantizar la fluidez de la información y del trabajo para hacer evolucionar el software hacia una solución que se la va entendiendo a medida que el proyecto avanza.

Los mundos mencionados del desarrollo de software son los definidos por las *metodologías tradicionales* como Waterfall y RUP, defensoras de la planificación detallada, la documentación extensa y los procesos formales; y las autodenominadas *metodologías ágiles* defensoras de los equipos auto-gestionados, la comunicación fluida, la construcción incremental y la planificación solo de lo es más cercano y se conoce.

¿Pero qué hay en medio? Parece que algo está faltando. Entre ambos mundos del desarrollo de software parece haber una zona de indefinición en la que caen muchos proyectos que terminan en grandes desvíos de tiempos y costos; claros síntomas de baja productividad.

Por un lado, es razonable pensar que es muy complicado hoy planificar un proyecto de varios meses, o varios años, asumiendo que los requerimientos no van a cambiar. La

dinámica actual de los negocios ya no permite esto. No parece posible cerrar etapas de proyectos para pasar a las que siguen sobre resultados firmes, algo seguramente va a cambiar y hay que poder reaccionar rápido. Es atractiva la idea de empezar a construir software que irá evolucionando con el paso de los ciclos y las entregas. El enfoque de las metodologías ágiles parece ser correcto en este aspecto.

Pero por otro lado, no parece posible hacerlo así desde el principio para todos los sistemas. No parece posible en todos los casos comenzar a construir sin definiciones y recorrer un camino que no se sabe hacia dónde llevará en términos de arquitectura y diseño del sistema, para luego hacer "refactoring". El costo de esta actividad puede ser tan alto que podría perderse el beneficio inicial de las entregas rápidas. Hay sistemas cuyo nivel de complejidad hace necesaria una etapa inicial de elaboración de la idea antes de comenzar la construcción, algún avance en arquitectura y diseño que permita iniciar la construcción con decisiones técnicas tomadas que aseguren un buen funcionamiento del sistema a largo plazo.

Dicho de otra forma, no parece ser posible en todos los casos usar métodos tradicionales puros, o métodos ágiles puros. Es necesario encontrar un intermedio que permita aprovechar la suma de los beneficios que ambos mundos prometen.

Asumiendo que la complejidad de los sistemas que se desarrollan en el mundo responden a una distribución normal (campana de Gauss) se podría decir que aparecerán a la izquierda de la campana un 15% aprox. de sistemas cuya baja complejidad y baja criticidad los hace candidatos firmes a la aplicación directa de metodologías ágiles bajo el concepto de comenzar a construir y luego refactorizar si fuera necesario. En el otro extremo de la campana habrá otro 15% de sistemas cuya complejidad y criticidad hacen imposible la aplicación directa de las metodologías ágiles ya que sus altos estándares de performance, robustez y calidad llevan indefectiblemente a etapas previas de ingeniería de requerimientos, definición de arquitectura y diseño que no pueden dejar de estar presentes para evitar fallas costosas y re-trabajo.

Pero queda el otro 70%, el centro de la campana, la mayoría de los sistemas del mundo, con niveles de criticidad y complejidad media, sobre los que perfectamente se pueden aprovechar las ventajas de la agilidad sobre una base de definiciones que arquitectura y diseño que facilite el camino y permita llegar a un resultado final óptimo con menor necesidad de refactoring.

Sobre la base de esta idea, en esta investigación se buscará una forma adecuada de unir las metodologías tradiciones y las metodologías ágiles en un intermedio lógico que permita aprovechar las ventajas que ambas tienen, y poner foco en sumar las buenas prácticas de cada una. El enfoque parece correcto, porque como se ha visto en capítulos anteriores, los factores humanos y los factores de proceso parecen ser los más determinantes para lograr buena productividad; y mientras los métodos ágiles ponen foco principalmente en los factores humanos, los métodos tradicionales lo hacen mayormente sobre los procesos.

Esta idea de que a las metodologías ágiles les falta algo para poder llevar adelante en forma exitosa proyectos de cierta complejidad, fue desarrollada también por Scott Ambler en su libro Disciplined Agile Delivery: A Practitioner´s Guide to Agile Software Delivery in the Enterprise[41]. Ambler propone ir más allá de Scrum y los métodos ágiles relacionados, completando la metodología con las cosas que a Scrum le faltan para poder ser un método completo que abarque todo el ciclo de vida de un proyecto. Ambler pone especial foco en una etapa inicial de formación del equipo y definiciones técnicas, y una etapa final de delivery del software en producción.

Según sus estudios, los equipos que hacen Scrum, gastan un promedio de 1 mes de trabajo al inicio de cada proyecto antes de poder iniciar la construcción, y otro mes al finalizar el proyecto para tareas de transición y puesta a punto del sistema en producción.

[41] Scott Ambler es un ingeniero de software canadiense, nacido en 1966. Estudió en la Universidad de Toronto, donde recibió sus títulos de grado y posgrado en Ciencias de la Computación y Ciencias de la Información. Se desempeñó en muchos roles como Business Architect, System Analyst, System Designer, Project Manager, Smalltalk programmer, Java programmer, y C++ programmer. También lideró el desarrollo de procesos de software como Agile Modeling (AM), Agile Data (AD), Enterprise Unified Process (EUP) y Agile Unified Process (AUP). Trabajando en IBM desarrolló el Disciplined Agile Delivery (DAD) con el que propone una forma de escalar los métodos ágiles a proyectos grandes, complejos y de misión crítica.

La propuesta de Scott Ambler, denominada DAD – Disciplined Agile Delivery, completa el método Scrum identificando 3 grandes etapas o fases cuyos nombres parecen tomados de las fases de RUP:

- **Inception**: es la etapa inicial que le faltaba a Scrum y donde se hacen las primeras definiciones técnicas que guiarán el resto del desarrollo, como una arquitectura y un diseño inicial. También ocurre en esta etapa la conformación del equipo de trabajo, cosa que los métodos ágiles en general dan por hecho.
- **Construction**: es la etapa de la construcción del software que usa métodos ágiles. Aquí con Scrum y XP se construye el software en ciclos cortos y mucha interacción con el cliente.
- **Transition**: la etapa final dedicada a poner el sistema a funcionar en producción, donde posiblemente haya que hacer actividades tampoco contempladas por Scrum y XP como instalar o actualizar hardware.

Cada una de estas tres grandes etapas tiene sus objetivos definidos para lograr un proceso completo, incorporando prácticas, actividades y objetivos que ayudan a visualizar un primer método combinado entre ágiles y tradicionales:

Figura 11: DAD - Etapas y Actividades

Inception	Construction	Transition
• Formar el equipo	• Producir el software en ciclos cortos	• Asegurar que la solución será usable
• Generar la visión común	• Recibir y priorizar los cambios	• Poner a funcionar el sistema
• Definir el Alcance preliminar	• Avanzar hacia el release final para producción	
• Identificar la estrategia técnica	• Mejorar la Calidad	
• Desarrollar el plan general	• Probar la arquitectura en forma temprana	
• Preparar ambiente de trabajo		
• Identificar Riesgos		

Fuente: http://stackoverflow.com + elaboración propia.

Otro aporte de DAD sobre la propuesta de Scrum, es una definición de roles más extensa, donde aparecen algunas funciones importantes que Scrum no considera. A su vez, los roles están clasificados en primarios y secundarios, siendo estos últimos de uso opcional según el tipo de proyecto y las necesidades:

Roles Primarios (Obligatorios):

- **Stakeholder**: es cualquier persona potencialmente afectada por el proyecto, no solo el usuario final.
- **Product Owner**: es la voz del cliente dentro del equipo, representa a la comunidad de los stakeholders.
- **Team member**: son los miembros del equipo e desarrollo que se van a enfocar en las actividades relativas a la constricción del software (análisis, diseño, estimaciones, programación, testing, etc.).
- **Team Lead**: es el líder del equipo en términos de aplicar correctamente las prácticas ágiles. Debe asegurar que el team tenga los recursos necesarios para hacer el trabajo, facilita la comunicación, soluciona problemas que frenan el desarrollo.
- **Architecture Owner**: Toma las decisiones de arquitectura del software y trabaja para crear, mantener y hacer evolucionar el diseño general del sistema.

Roles Secundarios (Opcionales, según las necesidades del proyecto):

- **Specialist**: especialista de algún tema en particular que sea necesario para complementar los conocimientos de los team members.
- **Domain Expert**: es un rol que en dominios complejos puede ayudar y complementar los conocimientos del Product Owner. Es un experto funcional.
- **Technical Expert**: un experto técnico que se incorpora al equipo solo si es necesario para aportar conocimiento sobre temas puntuales.
- **Independent Tester**: Aunque generalmente el testing lo hace el mismo equipo, en algunos casos de dominio o sistemas complejos puede ser importante que un equipo de testers trabaje en paralelo para asegurar calidad.
- **Integrator**: para sistemas complejos, uno o más integradores pueden ayudar a poner en funcionamiento el sistema completo a partir de sus sub-sistemas o módulos.

Estos roles complementan y completan las propuestas de los métodos ágiles, incorporando algunos roles que no estaban definidos pero que son importantes para proyectos de cierta complejidad. Por ejemplo, el rol del **Architecture Owner** que le da presencia e importancia a la definición y evolución de la arquitectura del sistema, o el rol del **Independent Tester** que otorga mayor presencia a la actividad de detección y remoción de los defectos.

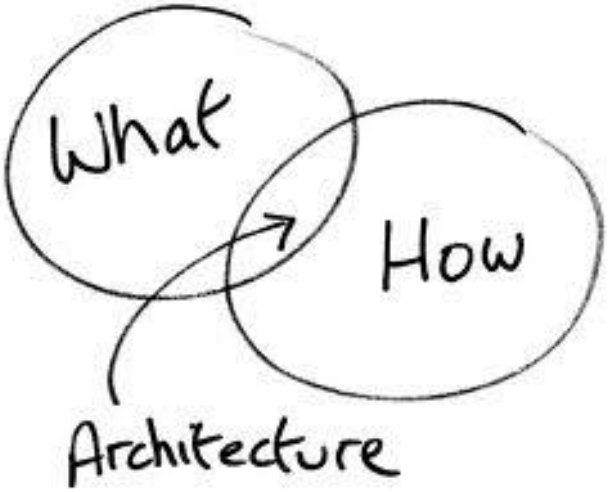
Esta propuesta presentada por Scott Ambler genera un marco de trabajo que, partiendo de las bases propuestas por los métodos ágiles, incorpora prácticas importantes del mundo tradicional, generando un marco de trabajo más completo y adecuado para proyectos de cierta complejidad.

En un segundo paso hacia la búsqueda y definición de una propuesta superadora, que aproveche las ventajas de los 2 mundos mencionados, aparece ACDM (Architecture Centric Development Method), una propuesta de Anthony J. Lattanze[42], un investigador de la Escuela de Ciencias de la Computación de la Universidad Carnegie Mellon. Anthony J. Lattanze describe en un paper publicado en 2005 un método de desarrollo de software cuya particularidad es la de estar centrado en la definición de la arquitectura, y que luego esta arquitectura es usada como fuente de las respuestas a todas las preguntas técnicas y de gestión del proyecto. La arquitectura en el centro postula Lattanze en su trabajo.

Según Lattanze, la arquitectura es la intersección donde los requerimientos (qué hay que hacer) se juntan con el espacio de la solución (cómo hacerlo). Es el primer esbozo de la solución técnica, es un diseño de alto nivel que permite visualizar su estructura principal, el particionamiento de un sistema en sus sub-sistemas o módulos, y cómo los módulos se relacionan entre ellos, las interfaces, para producir el funcionamiento deseado.

42 Anthony J. Lattanze es profesor y Director de carrera en la Universidad Carnegie Mellon y fundador de AJL Consulting. Especializado en el diseño de arquitecturas de sistemas. Previo a su trabajo en Carnegie Mellon desarrolló una extensa experiencia trabajando como ingeniero de software en varios programas de desarrollo y pruebas de vuelo de aviones de la Fuerza Aérea de los Estados Unidos. Luego del paper donde propone el método ACDM, en 2008 publicó el libro Architecting Software Intensive Systems: A Practitioners Guide.

Figura 12: ACDM - ¿Qué es la arquitectura?



Fuente: internet, http://stackoverflow.com

Otra definición clara de arquitectura es la que da Philip Lew, CEO de XBOSoft, una empresa basada en San Francisco, California[43]: "*Software architecture is the building blocks or components of software, their definition, and how those components are connected together, and their interactions*".

Queda claro que la arquitectura es un primer nivel del diseño del sistema que permite identificar los principales componentes, sus relaciones y responsabilidades. La arquitectura permite visualizar de antemano en qué medida el sistema podrá satisfacer los requisitos funcionales y también los no funcionales que son llamados **atributos de calidad** (escalabilidad, performance, portabilidad, etc.). El gran valor de la arquitectura es el de permitir al equipo de desarrollo avanzar sobre la construcción del sistema sobre una estructura definida que dé bastante certeza de que los requerimientos podrán ser satisfechos a largo plazo y con un nivel de refactoring controlado.

ACDM proponer dar a la arquitectura en desarrollo de software la misma importancia que se le da a la arquitectura en la industria de la construcción de edificios. En esa actividad los

[43] Definición publicada por el SEI – Software Engineering Institute junto con otras definiciones seleccionadas de arquitectura de software en la página titulada **Community Software Architecture Definitions** (http://www.sei.cmu.edu/architecture/start/glossary/community.cfm).

arquitectos participan del proyecto desde el inicio, modelando la idea hasta llegar a una representación gráfica del edificio a construir que satisface las expectativas de los stakeholders. A partir de la arquitectura es posible definir los materiales necesarios, las diferentes actividades que deben ser realizadas, es posible estimar tiempos y costos para definir el plan del proyecto y hasta permite determinar la composición y tamaño de los distintos equipos de trabajo que deberán participar del proyecto, las herramientas y maquinarias necesarias.

Esto es así, en la industria de la construcción, porque el producto es tangible, y todo el mundo tiene en claro que no es fácil ni barato deshacer lo hecho y/o cambiarlo para poder satisfacer necesidades no detectadas o nuevas expectativas. Resulta evidente la necesidad de modelar antes de comenzar a construir. A nadie se le ocurriría comenzar a construir un edificio sin antes tener planos aprobados por los diferentes interesados. No pasa lo mismo en el software, cuya intangibilidad esconde muchas veces la verdadera complejidad y costo de construirlo, y también de modificarlo. La percepción general es en muchos casos que desarrollar software es algo fácil, y que se puede entonces comenzar a construir, porque luego será fácil cambiar lo que haya que cambiar. Esta idea lleva a muchos proyectos a importantes atrasos y sobrecostos.

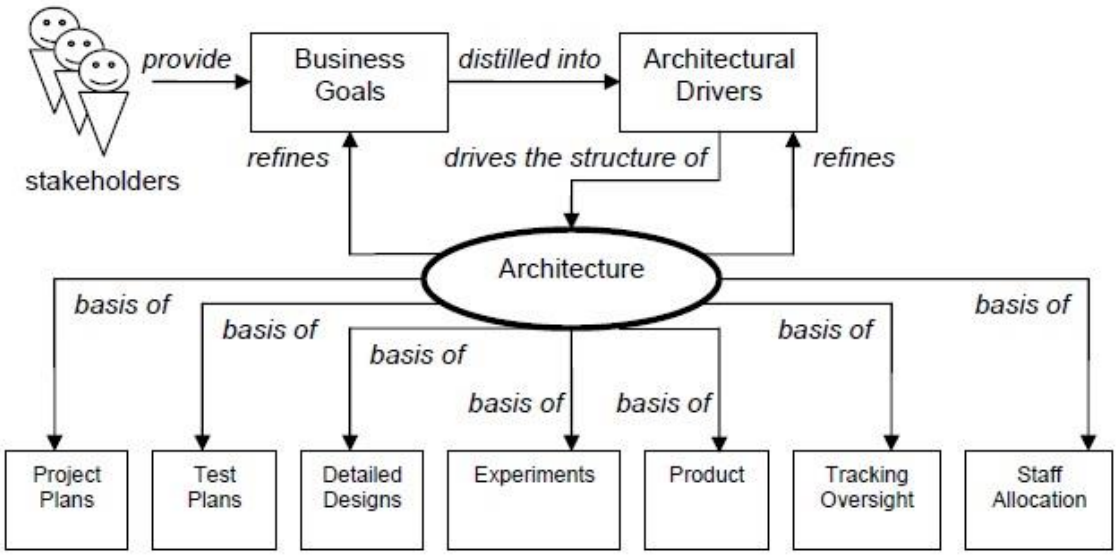
La arquitectura en software es la forma de garantizar que el sistema que será construido podrá en el futuro satisfacer plenamente no solo los requerimientos de funcionalidad solicitados, sino también los atributos de calidad como la escalabilidad, interoperabilidad, performance, mantenibilidad, y varios otros que son tan importantes como los funcionales si se tiene en cuenta que el sistema deberá funcionar por muchos años y deberá acompañar el crecimiento del negocio o de la actividad que sea.

Lattanze ve también la falencia que en este aspecto tienen los métodos ágiles, que empujan a los equipos de desarrollo a comenzar rápidamente la construcción del software, dejando que la arquitectura evolucione sola a medida que avanza la construcción. El problema en estos casos es que una arquitectura que emerge de la construcción muy posiblemente no sea la adecuada para satisfacer los atributos de calidad antes mencionados y el equipo de

proyecto estará obligado a un "refactoring" que podría ser muy costoso, o hasta imposible. Una arquitectura que emerge de la construcción es impredecible, y sus propiedades y atributos de calidad también lo serán.

Una ventaja interesante de este enfoque, es que poner a la arquitectura como el centro de la actividad del proyecto, no responde solo a las cuestiones técnicas de la construcción del software, sino que da información muy valiosa para determinar las actividades a realizar, los tiempos estimados, la conformación y organización adecuada de los equipos del proyecto y hasta los patrones de comunicación. Lattanze da un ejemplo interesante al mencionar que si 2 módulos del sistema tienen un alto nivel de integración e interacción, entonces los equipos que desarrollen esos módulos deberán tener también una comunicación muy fluida; en cambio los equipos que desarrollen módulos con bajo nivel de interacción seguramente tendrán poca comunicación. La arquitectura permite entonces organizar los equipos de desarrollo, asignar el trabajo de la forma más adecuada, y hasta definir las ubicaciones físicas de los equipos para facilitar la interacción necesaria.

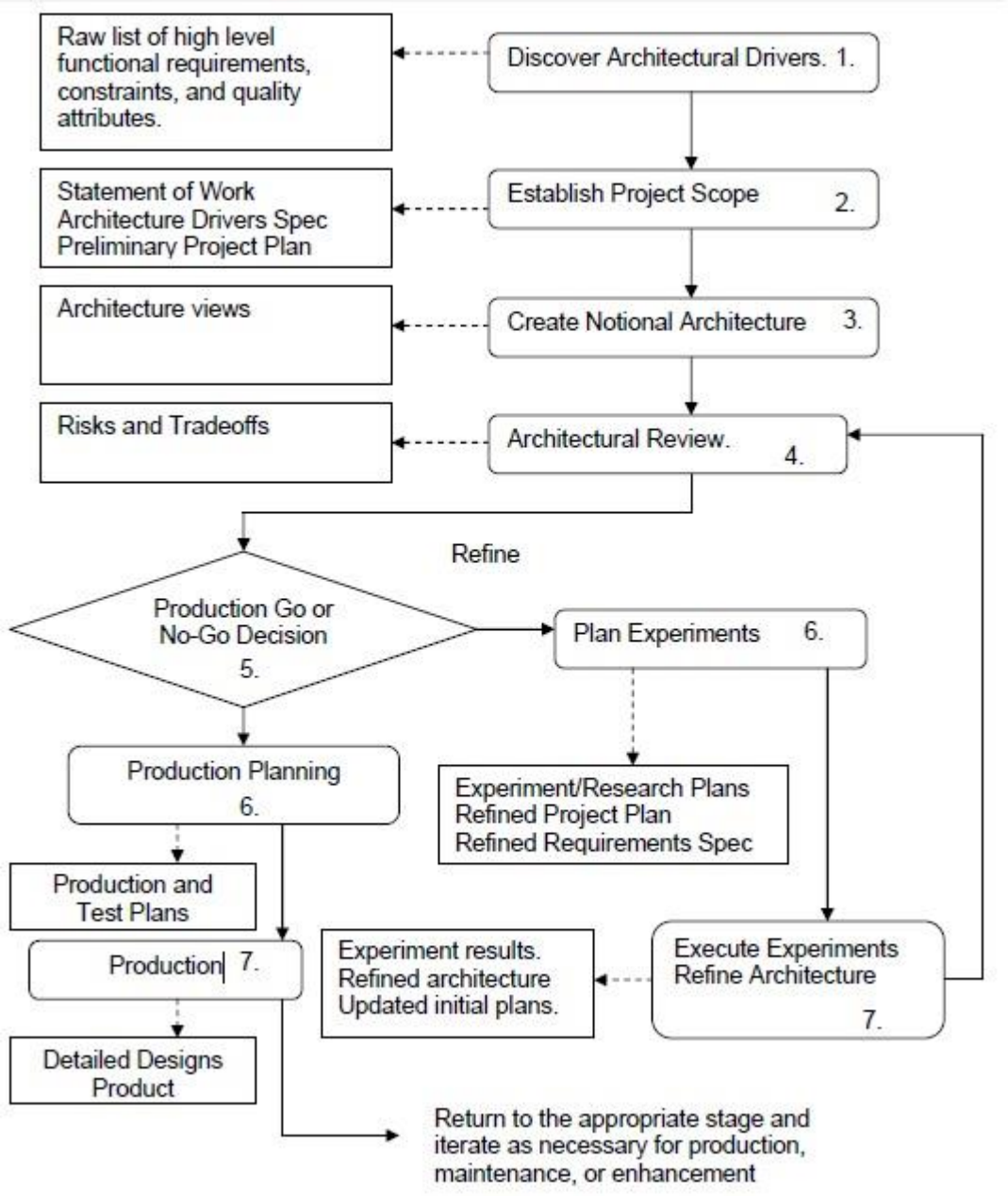
Figura 13: ACDM - La arquitectura en el centro



Fuente: A. J. Lattanze, *The Architecture Centric Development Method*, USA, 2005, *p.* 56.

ACDM propone un método dividido en 7 etapas o "stages" que llevan primero al equipo de desarrollo a realizar las grandes definiciones y la arquitectura, para pasar luego a la construcción recién cuando se considera que la arquitectura está madura o "ready for production".

Figura 14: ACDM – Las 7 Etapas



Fuente: A. J. Lattanze, *The Architecture Centric Development Method*, USA, 2005, *p.* 9.

Los primeros 4 stages están relacionados a la captura de los requerimientos de los clientes, identificar los atributos de calidad los aspectos que habrá que tener en cuenta en la arquitectura para poder satisfacerlos. A partir de esta información de elabora la definición del alcance, la planificación preliminar del proyecto y la arquitectura propiamente dicha:

- **Stage 1: Discover Architectural Drivers**. Reuniones con el cliente para descubrir y documentar requerimientos, restricciones, atributos de calidad y los drivers de la arquitectura.
- **Stage 2: Establish Project Scope**. Refinar y documentar la información obtenida en la etapa anterior. Crear un Statement of Work (SOW) y una planificación preliminar.
- **Stage 3: Create Notional Architecture**. Crear la arquitectura inicial con varias vistas del sistema. Hay métodos específicos para desarrollar y documentar arquitecturas como Quality Attribute Workshop (QAW), Architecture Tradeoff Analysis Method (ATAM), o Attribute Driven Design (ADD).
- **Stage 4: Architectural Review**. Revisar la arquitectura alcanzada para poder refinarla y mejorarla; descubrir posibles problemas, limitaciones o riesgos a trabajar.

El Stage 5 es central para ACDM ya que es aquí donde se decide si la arquitectura está lista para pasar a la construcción del sistema o se vuelve a stages anteriores para refinarla:

- **Stage 5: Production Go/No-Go**. Listar y priorizar los problemas, limitaciones y riesgos encontrados en la revisión y decidir si la arquitectura alcanzada está lista para iniciar la construcción.

A partir de aquí se abren 2 caminos posibles a seguir, Refine (No-Go) para cuando la arquitectura no está lista y debe ser refinada, y Production (Go) para cuando si está lista y comienza la construcción:

Camino del Refine (No-Go) – La arquitectura necesita ser refinada

- **Stage 6: Experiment Planning**. Se planifica la realización de pruebas (experimentos), prototipos, ensayos, o cualquier cosa que permita entender mejor los problemas y riesgos encontrados y como solucionarlos.

- **Stage 7: Experiment Execution and Architecture Refinement**. Se realizan los experimentos y se refina la arquitectura según los resultados obtenidos. Aquí se vuelve al Stage 4 para retomar el camino.

Camino de Production (Go) – La arquitectura está lista para iniciar la construcción

- **Stage 6: Production Planning**. Se crea el plan detallado para la construcción del sistema basado en los elementos o módulos que la arquitectura define. A cada elemento se le asigna un "owner" o responsable por su construcción. Se planifica para cada módulo las actividades de diseño, construcción, revisiones, testing, etc.
- **Stage 7: Production**. Los equipos ejecutan el plan de construcción de cada módulo, esta etapa se realiza bajo los lineamientos que la arquitectura marca asegurando que se cumplen las grandes decisiones técnicas. Esta parte incluye también las pruebas de cada módulo, pruebas de integración del sistema y todo lo que se haya planificado en el Stage anterior. Si la planificación es iterativa, en este punto se vuelve al Stage 1 para comenzar nuevamente.

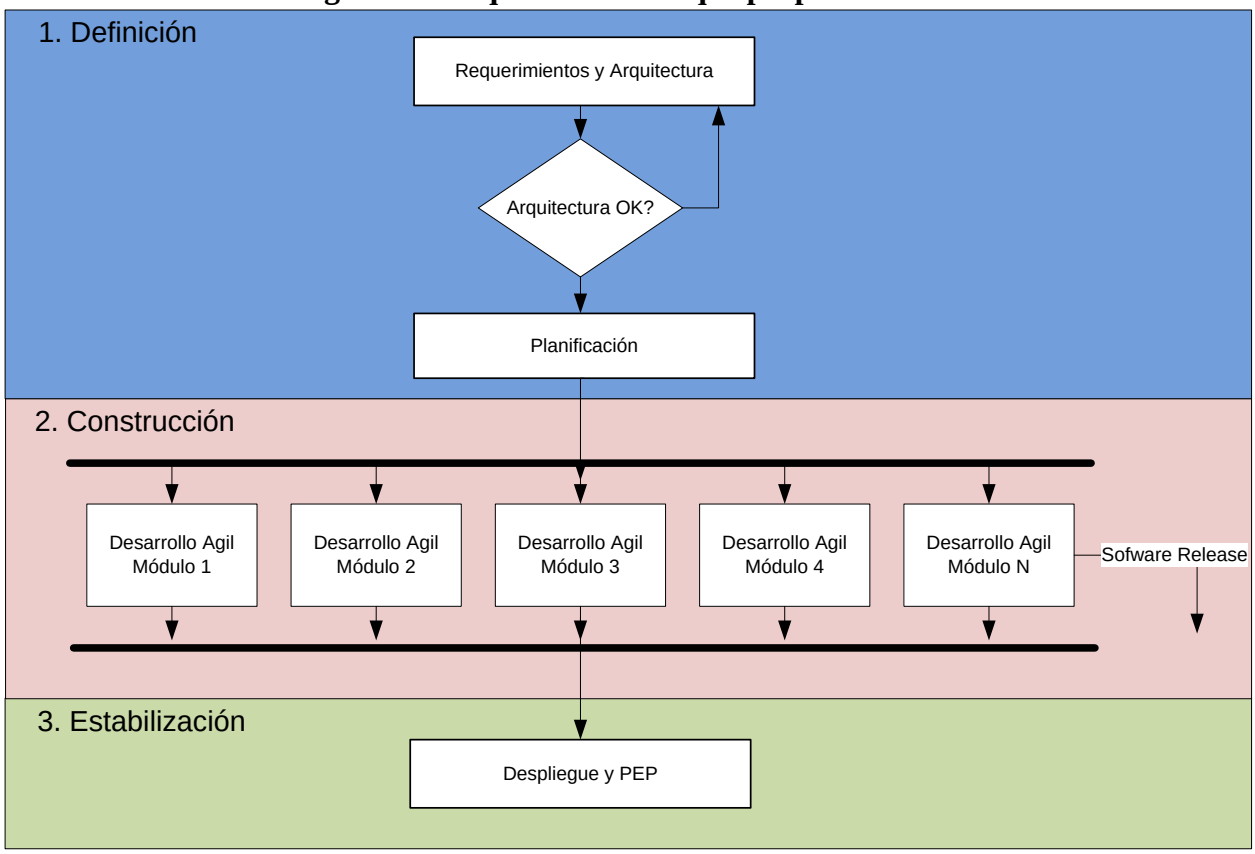
Las actividades de producción del Stage 7 pueden ser perfectamente realizadas bajo los postulados de los métodos ágiles como Scrum o XP, donde cada módulo o elemento puede ser construido en incrementos cortos que permitan alimentar una actividad de integración temprana, pero todo sobre la base de una definición de arquitectura que aumenta las posibilidades del sistema de satisfacer los atributos de calidad requeridos en el corto, mediano y largo plazo.

Es claro que los arquitectos no van a poder diseñar una arquitectura que soporte cualquier cambio que vaya a ocurrir en el futuro o cualquier necesidad que pueda aparecer, pero invertir tiempo en las etapas iniciales para desarrollar una arquitectura de base, seguramente permitirá demorar, reducir y hasta posiblemente eliminar la necesidad de un rediseño, aumentando el ciclo de vida útil del sistema y maximizando su retorno económico.

Ambas metodologías descriptas, DAD y ACDM dan la idea de que es posible unir los mundos de los métodos tradicionales y ágiles para aprovechar las ventajas de ambos en un enfoque que permita a las empresas asegurarse que las mejores prácticas de la industria estarán presentes en sus procesos y con ellas ir en búsqueda de una buena productividad.

El siguiente es un enfoque propuesto con este objetivo, aprovechar las buenas prácticas de los métodos ágiles en cuanto a gestión de los requerimientos, entrega rápida de valor al cliente, y el funcionamiento de los equipos compactos y auto-gestionados, sobre el paraguas de una etapa previa de definición donde se logra una arquitectura que guía el resto del proyecto y que asegura el cumplimiento de los atributos de calidad requeridos por el cliente:

Figura 15: Esquema del enfoque propuesto



Fuente: elaboración propia.

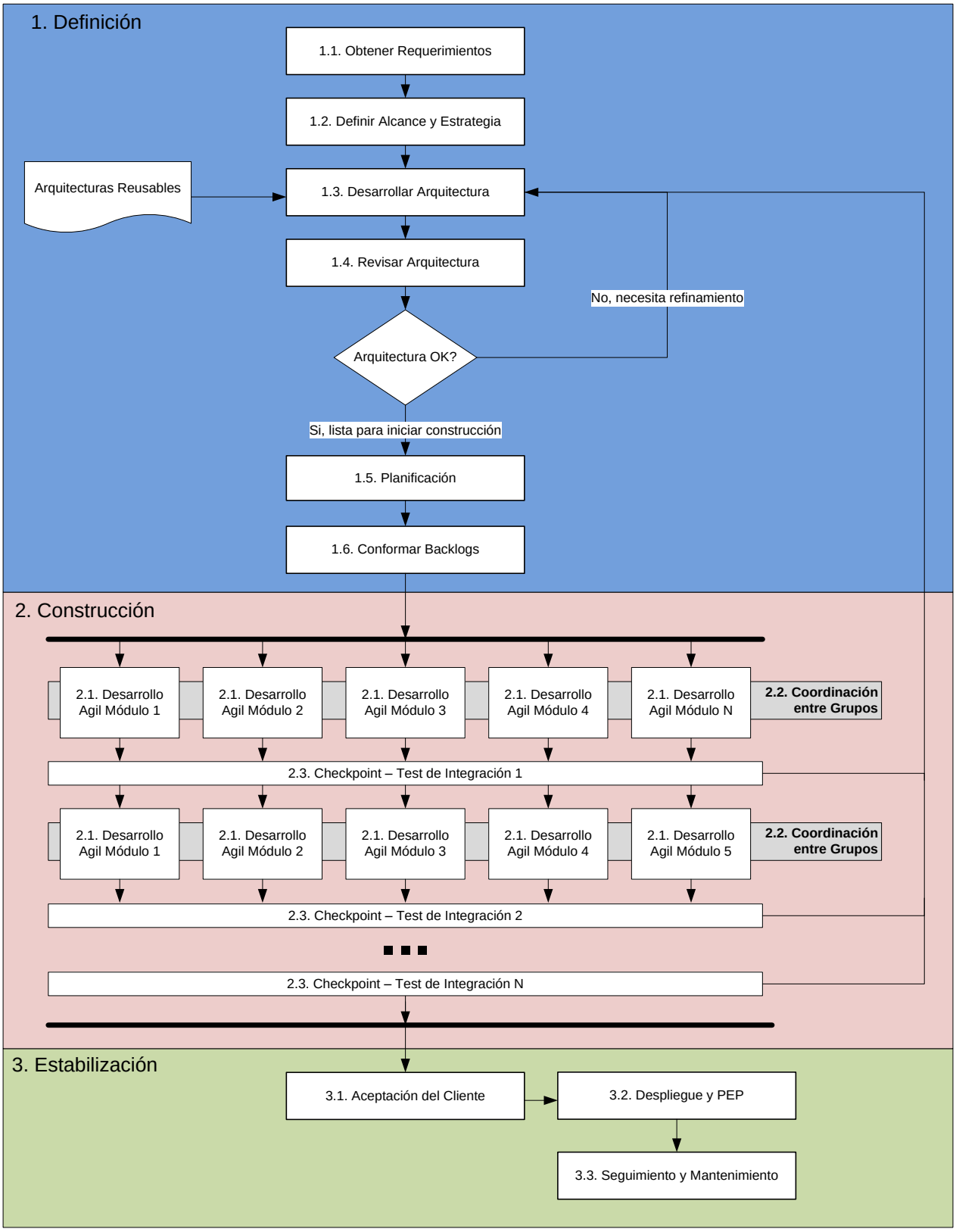
El enfoque está principalmente dividido en tres etapas llamadas **Definición**, **Construcción** y **Estabilización**:

- La etapa de **Definición** tiene el objetivo general de tomar los requerimientos del cliente y desarrollar una arquitectura que permita luego la planificación y asignación del trabajo. La arquitectura es revisada y no se pasa a la etapa siguiente hasta no obtener su aprobación. En base a la arquitectura se realiza la planificación, asignando la construcción de los componentes a diferentes equipos de desarrollo.
- La etapa de **Construcción** es puro desarrollo ágil siguiendo métodos como Scrum y XP donde los equipos de desarrollo construyen los módulos que la arquitectura define. Esto le da a cada grupo de desarrollo la libertad de trabajar los módulos según su mejor criterio y experiencia pero sin salirse de lo que la arquitectura define. Cada grupo hará diseño, codificación y refactoring según sea necesario. Es importante en esta etapa la comunicación y sincronización entre los equipos de desarrollo, especialmente entre aquellos que trabajan en módulos que están fuertemente relacionados en la arquitectura.
- La etapa de **Estabilización** es para la puesta en producción final del sistema y su estabilización hasta poder cerrar el proyecto y/o pasar a mantenimiento. Se apunta a que esta etapa sea lo más corta posible, cosa que podrá ser lograda si el trabajo en las etapas anteriores fue bien hecho, ya que será necesaria poca estabilización. El sistema debería ya estar trabajando bien desde las entregas previas realizadas.

Lo que se describe en esta sección no pretender ser una metodología completa de desarrollo de software, ni tampoco un proceso detallado con sus actividades, entradas y salidas. Es simplemente un enfoque que combina pasos y procedimientos existentes que ya están descriptos en sus métodos originales. Este enfoque podría ser usado por una empresa como forma de organizar sus proyectos de software, recurriendo al detalle de los métodos que lo componen como DAD, ACDM, SCRUM, XP, etc.

La siguiente figura muestra una visión más detallada de este enfoque, abriendo las tres grandes etapas en sus actividades internas y relaciones:

Figura 16: Detalle del enfoque propuesto



Fuente: elaboración propia.

La etapa de **Definición**, tiene el objetivo principal de lograr un nivel de definiciones suficientemente maduro como para ingresar a la Construcción con las grandes decisiones técnicas y de gestión del proyecto tomadas, para esto se llevan a cabo las siguientes 8 actividades:

1.1. Obtener Requerimientos
Descripción de la Actividad
Mediante reuniones con el cliente se obtiene un buen entendimiento de lo que el sistema debe hacer, los aspectos funcionales y técnicos que deben ser satisfechos, las limitaciones, los objetivos de negocio que el sistema ayudará a lograr, etc.
Aspectos Principales
La información a obtener debe ser la suficiente para poder avanzar en el desarrollo de la arquitectura y la planificación. El contacto y el involucramiento del cliente deberán ser mantenidos durante todo el proyecto por lo que no es necesario obtener en este punto "toda" la información ni todo el detalle, solo la información necesaria para avanzar y que luego podrá ser refinada en etapas posteriores.

1.2. Definir Alcance y Estrategia
Descripción de la Actividad
Este es un trabajo de bajar en limpio la información obtenida en la actividad anterior para clarificar el alcance del proyecto, el entendimiento de la funcionalidad, priorizar los requerimientos y definir una estrategia técnica y de gestión del proyecto que será utilizada.
Aspectos Principales
Es importante en esta etapa que aparezcan y se documenten los driver principales del proyecto, ya sean funcionales o técnicos, y las restricciones y limitaciones, en las que habrá que poner foco luego cuando se diseñe la arquitectura del sistema para asegurar que esta los contempla.

1.3. Desarrollar Arquitectura
Descripción de la Actividad
Con la información anterior se desarrolla la arquitectura del sistema, considerando que esta no será la arquitectura final, sino que se deberá llegar a un nivel de definición suficiente como para avanzar y luego la arquitectura podrá ser refinada. La arquitectura planteada debe contener al menos la identificación de los componentes o módulos del sistema, la responsabilidad o funciones de cada uno y la definición de las interfaces que cada módulo usará para comunicarse con los demás. Pueden desarrollarse alternativas de arquitectura si se presentan situaciones de tradeoff entre funciones o atributos de calidad.
Aspectos Principales
Los requerimientos funcionales y los atributos de calidad como escalabilidad y performance deben estar considerados. En esta actividad juega un papel clave un concepto que es fundamental para la productividad, es el concepto del **reúso de componentes**, donde la arquitectura es primordial como la herramienta que permite determinar cuáles patrones de arquitectura o componentes poder ser reusados de proyectos anteriores, o cuáles pueden ser construidos de forma genérica y encapsulada para reúso.

1.4. Revisar Arquitectura
Descripción de la Actividad
La arquitectura desarrollada es presentada y revisada por el equipo de proyecto para determinar en qué medida está lista para iniciar la construcción de los módulos del sistema. Puede ser presentada también al cliente si este tiene los skills necesarios para evaluarla. Se explica, discute y evalúa cómo cada funcionalidad y atributo de calidad son atendidos o no por la arquitectura, de evalúan las posibles alternativas y se identifica y categoriza la deuda técnica generada. Según lo que se defina es posible continuar con la planificación y construcción, o volver a la actividad de definición de la arquitectura para refinarla.
Aspectos Principales
Se debe tener en cuenta que la arquitectura que se libere para iniciar la construcción no será la final y definitiva. La arquitectura es también parte de la filosofía iterativa de estos métodos y podría ser refinada más adelante. Pero se debe evaluar que el nivel de madurez alcanzado sea adecuado para evitar a futuro grandes re-trabajos.

1.5. Planificación
Descripción de la Actividad
Con la información de la arquitectura, el alcance del proyecto y la estrategia elaboradas antes de realiza la planificación del proyecto. Básicamente se define qué se debe hacer, quién lo va a hacer, cuando, con qué herramientas y recursos; y todo con la información que brinda la arquitectura. La planificación debe incluir las actividades necesarias para saldar la deuda técnica que se decida tomar en las primeras entregas.
Aspectos Principales
La arquitectura es el primer nivel de una lista de entregables tipo WBS ya que da la lista de componentes a construir; a partir de allí y teniendo en cuenta las cuestiones técnicas, recursos disponibles, experiencia y conocimientos de las personas y grupos, se asignan los componentes a los grupos, considerando aspectos importantes como que módulos muy integrados en la arquitectura sean desarrollados por el mismo grupo o por grupos que puedan interactuar fácilmente.

1.6. Conformar Backlogs
Descripción de la Actividad
Como una preparación para el inicio de la construcción de cada componente usando métodos ágiles, se construye el backlog del producto o sistema y el backlog individual de cada componente que será seguido por cada equipo de desarrollo. El backlog del producto debe identificar para cada funcionalidad qué componentes están involucrados. El backlog de cada componente debe identificar a qué funcionalidad del producto corresponde cada función del componente, de forma de permitir un avance coordinado.
Aspectos Principales
Es importante mantener la relación y las prioridades sincronizadas entre los backlogs del producto y de los componentes para posibilitar que haya puntos intermedios de control, pruebas de integración y hasta entregas al cliente final donde todos los módulos implementaron funcionalidades que permiten completar las funcionalidades más prioritarias del sistema completo. La gestión posterior de los backlogs y avance de los desarrollos debe ser sincronizada en este aspecto.

La etapa de **Construcción**, tiene el objetivo principal de construir los módulos aprovechando las ventajas de los métodos ágiles, pero será necesario un nivel de coordinación entre los equipos que no está mencionado por los métodos Scrum o XP. Esto es especialmente importante ya que los grupos de desarrollo no estarán construyendo cada uno un sistema independiente, sino uno o más componentes de un sistema más grande que deben luego integrarse y funcionar. El cliente tiene que estar involucrado en esta etapa, aportando información a los grupos de desarrollo y participando de las pruebas de integración:

2.1. Desarrollo Ágil Módulo 1, 2, 3, … N
Descripción de la Actividad
Esta actividad es pura construcción ágil, siguiendo los preceptos de los métodos más populares. Los equipos de desarrollo toman los requerimientos del backlog y los construyen en ciclos de desarrollo cortos. Se realizan las reuniones de seguimiento diarias, estimaciones por consenso, auto-asignación de actividades, codificación por pares, etc.
Aspectos Principales
Cada grupo tendrá asignado uno o más componentes del sistema a desarrollar y deberá seguir el o los backlogs correspondientes para avanzar en la implementación. Cada ciclo o sprint debería resultar en una versión funcional del o los módulos que puedan ser entregados a una actividad de testing de integración del sistema completo. Los límites en cuanto a la definición técnica están dados por la arquitectura ya que el grupo deberá respetar las especificaciones de funcionalidad, técnicas y las interfaces que se definieron. Dentro del módulo el equipo tiene total libertad para diseñar, codificar y refactorizar como sea necesario. Se usarán métodos ágiles y sus prácticas como Scrum, XP, Unit Test, TDD, Pair Programming, refactoring, etc.

2.2. Coordinación entre Grupos
Descripción de la Actividad
Es una actividad que se realiza en forma simultánea al desarrollo ágil y donde los equipos toman contacto entre ellos para informar su avance, discutir y solucionar problemas y mantener un nivel de coordinación y sincronismo que permitan llegar a puntos donde todos

los módulos puedan ser puestos a funcionar juntos para probar su integración.
Aspectos Principales
Así como las reuniones diarias de cada equipo sirven para mantener una comunicación fluida y destrabar situaciones de demoras o problemas, esta actividad persigue el mismo objetivo para mantener coordinados a todos los equipos entre sí. Podría tratarse de una reunión corta diaria o cada 2 días, donde, después de la reunión diaria de cada equipo un miembro (preferentemente el Líder) asiste a la reunión de coordinación. En esta reunión se comparte el avance de cada equipo, las funcionalidades implementadas y las próximas a implementar, resultados de pruebas de integración, problemas encontrados, necesidades de asistencia entre los equipos y todo lo necesario para mantener a los equipos en avance coordinado. Deben participar los responsables de la Arquitectura para estar al tanto de cualquier necesidad de ajuste o mejoras.

2.3. Checkpoint – Test de Integración 1
Descripción de la Actividad
Al final de cada ciclo, o cada N ciclos, todos los equipos entregan sus módulos para una prueba coordinada de integración donde el sistema es probado como un todo. De esta actividad pueden surgir problemas a solucionar o funcionalidades que deban ser cambiados o ajustadas en determinados módulos. Cada una de estas pruebas de integración podría resultar en un reléase del sistema potencialmente entregable al cliente y usable por este último.
Aspectos Principales
Esta actividad es esencialmente una actividad de testing, con el objetivo detectar y remover lo antes posible las fallas que se generen y que superen a las actividades propias de cada equipo como el Unit Test. Estará principalmente enfocada en pruebas de integración, pero podrá detectar cualquier tipo de fallas. Se contempla que en esta actividad se hagan varios tipos de pruebas: funcionales, de seguridad, de carga y stress, de redundancia, etc.

La etapa de **Estabilización** es para la puesta en producción final del sistema y su estabilización, las actividades de esta etapa tienden a cerrar todos los pendientes que puedan quedar, obtener la aceptación final del cliente y pasar a la etapa de operación del

sistema en régimen. Es muy posible que el sistema ya esté operando en producción desde las entregas anteriores de la etapa de Construcción:

3.1. Aceptación del Cliente
Descripción de la Actividad
Luego de las etapas de Construcción en las que el cliente participó y recibió varios releases del sistema, esta actividad final de pruebas de aceptación donde se busca la aceptación final de conformidad del cliente para el cierre del proyecto y los contratos. Las pruebas podrían realizarse siguiendo un protocolo de pruebas acordado previamente con el cliente.
Aspectos Principales
Se realizarán pruebas generales de funcionalidad del sistema y de los atributos de calidad, las que podrían desembocar en necesidad de ajustes o correcciones que alimentarán un nuevo ciclo de desarrollo. Así como las pruebas de integración de la etapa de Construcción, estas pruebas podrían incluir aspectos funcionales, o relacionados a los atributos de calidad del sistema como escalabilidad, performance, alta disponibilidad, etc.

3.2. Despliegue y PEP
Descripción de la Actividad
Se trata del despliegue final del sistema sobre los equipos destinados al uso productivo. Si el sistema ya está en producción producto de entregas anteriores, este despliegue será solo incremental para incorporar las funcionalidades y cambios pendientes de los últimos ciclos de desarrollo.
Aspectos Principales
A pesar de ser una actividad propia de la etapa de Estabilización, podría haber sido realizada antes en la etapa de Construcción como parte de las entregas intermedias generadas por los ciclos de desarrollo.

3.3. Seguimiento y Mantenimiento
Descripción de la Actividad
Toda aquella actividad necesaria para mantener el sistema en correcto funcionamiento según las obligaciones acordadas con el cliente (monitoreo, mantenimientos preventivos, soporte, capacitación, backups, etc.).
Aspectos Principales
En esta etapa suele verse reflejada la calidad del trabajo realizado en etapas anteriores y la cantidad y el tipo de deuda técnica generada durante el proyecto. Los sistemas con un alto costo de mantenimiento, difíciles de operar, con actividades preventivas y/o correctivas frecuentes, suelen ser el resultado de proyectos de desarrollo donde no se ha puesto especial foco en los temas relacionados a los atributos de calidad, la deuda técnica, el reúso de componentes y otros.

Habiendo finalizado ya la descripción de las etapas y actividades de este enfoque para el desarrollo de software, es importante describir brevemente a los roles intervinientes. La definición que respecto de roles realiza Scott Ambler en su propuesta llamada DAD (Disciplined Agile Delivery) parece adecuada y completa para cubrir todas las funciones y responsabilidades necesarias en un proyecto de desarrollo de software exitoso. Esta propuesta parte de la lista original de roles de Scrum, y la completa con nuevos roles para asegurar el enfoque disciplinado del proceso y garantizar el éxito y el cumplimiento de los objetivos las etapas nuevas llamadas **Inception** y **Transition** con las que enmarca al desarrollo ágil. Sin embargo, es importante hacer hincapié en aspectos fundamentales de algunos de los roles principales:

- **Stakeholder & Product Owner**: estos roles deben asegurar la presencia e involucramiento del cliente en todas las etapas del ciclo de desarrollo, incluso en plena construcción. Es importante contar siempre con información del cliente respecto de las funcionalidades y sus expectativas, como así también de su validación respecto de que el equipo está logrando materializar su visión de la solución. El Product Owner podría ser una persona del cliente, o un grupo de

personas del cliente, a modo de comité. En su defecto una persona de la organización con contacto muy fluido con el cliente como para poder representarlo.

- **Architecture Owner**: esta función es central para poder aprovechar las ventajas de ACDM, la arquitectura en el centro. Esto supone que la arquitectura es de una importancia tal para el equipo de proyecto que un especialista o un grupo de especialistas debe estar a cargo de su definición y evolución. Esta persona o grupo de personas, además debe estar presente e involucrado en todas las demás etapas y actividades para lograr que la arquitectura sea una buena base para la solución, que incluya y satisfaga todos los requisitos funcionales y técnicos, y que luego en la construcción, lo que efectivamente se construya siga y respete los lineamientos de la arquitectura definida.
- **Team Lead**: es el Scrum Master de Scrum con foco en la organización y dirección del equipo hacia un uso correcto de la metodología y los recursos. Pero en este caso tiene la responsabilidad adicional de mantener a su equipo coordinado con los otros equipos de desarrollo. Es muy importante su función en buscar acuerdos técnicos con los otros equipos, participar de las reuniones de coordinación, ayudar y pedir ayuda cuando sea necesario, y mantener el backlog de su equipo coordinado con los otros backlogs y alineado con las necesidades del proyecto.
- **Independent Tester & Integrators**: Serán los responsables de armar y ejecutar un plan de pruebas que pueda ser ejecutado en cada etapa de test de integración. Se supone que los equipos de desarrollo pondrán foco en la detección y solución de fallas de cada módulo, pero queda bajo la responsabilidad de estos roles el testing de integración donde podrían aparecer nuevas fallas ya que los módulos son puestos a trabajar juntos por primera vez luego de ser creados o modificados. Deben estar en línea con las entregas y siempre listos a desplegar y probar el sistema integrado en cualquier ambiente.

En el desarrollo de este enfoque se han tenido en cuenta muchos de los aspectos que afectan la productividad y que se fueron detectando y explicando en capítulos anteriores. Las etapas y pasos propuestos están diseñados para que en su conjunto maximicen las prácticas que se listan a continuación y que quedan para una revisión y conclusión en el capítulo final:

- Foco en las etapas iniciales donde se hace la Construcción Conceptual.
- Foco en la Arquitectura y en los Atributos de Calidad.
- Enfoque Iterativo de las definiciones y de la construcción.
- Gestión de la Deuda Técnica.
- Detección Temprana de los Defectos.
- Involucramiento del Cliente en todas las etapas y actividades.
- Definición y seguimiento de un proceso, actividad disciplinada.
- Reúso de Componentes, desde la Arquitectura hasta el código.
- Foco en el entrenamiento y la motivación de las personas, espacio para decidir, crear, exponer su talento y experiencia.
- Equipos chicos y auto-gestionados.
- Timeboxing, tiempos cortos asignados, foco en las entregas frecuentes.
- Lecciones Aprendidas, mejora de los procesos por parte del mismo equipo.
- Alta frecuencia de seguimiento del proyecto (reuniones diarias o cada 2 días).
- Propagación de la información y del conocimiento.
- Collective Code Ownership.
- Los 7 principios del Lean Software Development.

CAPITULO 8. CONCLUSIONES Y RECOMENDACIONES

Llegó la hora de resumir todo lo investigado y descubierto, en unas pocas páginas que le den a este trabajo un cierre, una conclusión, un aporte a quién vaya a leerlo en el futuro, preocupado por estos temas. La productividad general en actividades de desarrollo de software no está en su nivel óptimo y aún puede ser mejorada[44]. Esta es la hipótesis planteada, y a lo largo de los capítulos este trabajo fue descubriendo y analizando información que sustenta esta afirmación y demuestra la situación:

"El costo del software ha sido duramente impactado por pobre calidad, seguridad marginal y otros problemas crónicos" (Jones Capers, 2008, *p*. 568).

"La productividad real, que el individuo o grupo logra, raramente es igual a la potencial ya que los individuos y grupos fallan al hacer el mejor uso posible de los recursos disponibles" (Bianca Velázquez Rodríguez, 2009, *p*. 1).

"No hay una sola técnica de desarrollo, tecnología, o management que prometa por si solo una mejora de un orden de magnitud en productividad, confiabilidad y simplicidad" (Frederick Brooks, 1996, *p*. 1).

"*Hay mucho que hacer acerca de cómo mejorar la productividad, e incluso mejorar la productividad por un factor de 2 hará una diferencia significativa para la mayoría de las organizaciones*" (Barry Boehm, 1987, *p*. 1).

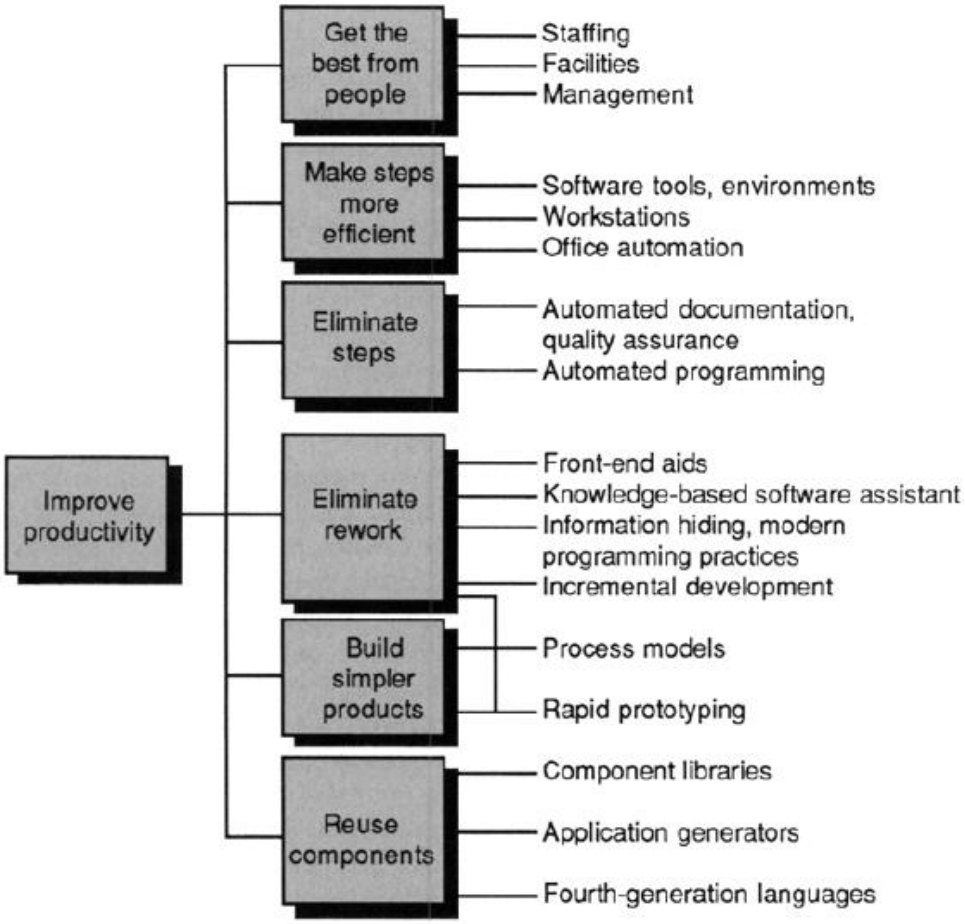
Las respuestas de los colegas de la industria del software, dan también cuenta de esta realidad, ya que indican que al menos el 50% de las personas entrevistadas piensan que la productividad en sus empresas puede ser mejorada. Esta realidad también surge como una de las conclusiones de la entrevista realizada a un referente de la industria nacional.

[44] Es la hipótesis planteada para este trabajo (*p*. 10).

Una de las cosas que queda clara es que producir software con buenos niveles de productividad y calidad, no es tarea fácil. Hay muchos factores que pueden afectar la productividad y que es necesario atenderlos a todos, o al menos a la gran mayoría, para lograr buenos resultados.

Barry Boehm (1987) plantea una idea interesante que llamó: "*The Software Productivity Improvement Opportunity Tree*". Es una apertura en forma de cuadro de los principales factores que él considera como fuentes de ahorro de costos y mejora de la productividad. Esta apertura jerárquica ayuda a entender y visualizar los puntos de mejora que según Boehm pueden hacer la diferencia, y coincide con la mayoría de los trabajos que buscan identificar los factores que afectan la productividad:

Figura 17 - El árbol de la oportunidad para la mejora de Barry Boehm



Fuente: Barry Boehm, *Improving Software Productivity*, USA, 1987, *p* 13.

Partiendo de la base de que **producir software es transformar conocimiento**[45], una actividad netamente humana, ayudada o asistida por computadoras, pero principalmente humana, el talento, la experiencia y la motivación de las personas resultan indispensables y no son fáciles de obtener. Luego siguen los procesos, la tecnología, y otros factores que acompañan, pero el factor humano es el principal; con su infinita capacidad e infinito potencial, pero también con gran complejidad para ser gestionado.

Se elija la metodología que se elija, hay que tener en cuenta estos factores, como también hay que tener en claro que no hay ninguna metodología que resulte ser perfecta ni mucho menos. No hay metodología que por sí sola asegure mediante su aplicación completa un alto porcentaje de probabilidad de obtener muy buenos resultados. Lo que si hay son diferentes organizaciones, grupos humanos, contextos, realidades de negocio, culturas, motivaciones, dominios de aplicación, y muchas otras cosas que se combinan en cada proyecto de desarrollo de software y crean un contexto único. Es por eso que lograr una buena productividad será en gran medida producto de la habilidad de las personas, especialmente los managers, en crear entornos de trabajo donde los temas que se han visto se tengan seriamente en cuenta y donde las buenas prácticas se pongan efectivamente en práctica.

A continuación un resumen, un checklist para el manager de un proyecto o grupo de desarrollo de software. Lo que no puede dejar de considerar para lograr buena productividad, o para mejorar la productividad actual. Esta lista de conceptos, ideas, enfoques y técnicas tiene en cuenta todo lo que se ha investigado hasta ahora en este trabajo, los descubrimientos y recomendaciones de los autores citados, y lo que las diferentes metodologías proponen:

Foco en las etapas iniciales donde se hace la Construcción Conceptual.

Lo dijo Brooks (1986) en su famoso paper: "*La parte dura de hacer software es la especificación, diseño y testing de la construcción conceptual, no el trabajo de representarlo y testear la fidelidad de la representación*". También lo mencionan

[45] Uno de los conceptos tomados de la entrevista realizada como parte de esta tesis. Ver detalle en el Anexo 3.

Blackburn y Scudder (1996) en lo que llamaron La Paradoja del Tiempo: "*Mejorar el tiempo total del proyecto no implica reducir el tiempo de todas las etapas, sino, por el contrario, dedicar más tiempo a algunas, especialmente las iniciales*". Es importante invertir algo de tiempo al inicio del proyecto para lograr un buen entendimiento de lo que hay que hacer y cómo se lo va a hacer. Luego se puede iniciar la construcción. También es vital considerar la importancia de quienes van a llevar adelante estas primeras etapas, Brooks (1986) se refiere a este tema en su paper: "*Los buenos diseñadores son tan importantes como los buenos managers. Ellos bajan los grandes errores conceptuales*".

Foco en la Arquitectura y en los Atributos de Calidad.

La definición de la arquitectura de un sistema es una buena base para asegurar que lo que se va a hacer va a poder satisfacer las necesidades funcionales y los atributos de calidad a largo plazo. Iniciar la construcción sobre la base de una buena arquitectura resultará en un sistema más robusto, que podrá ser utilizado por mucho tiempo con bajo costo de mantenimiento, mejorando considerablemente el retorno de la inversión.

Enfoque Iterativo de las Definiciones y de la Construcción.

No es posible hoy lograr una definición completa de lo que hay que hacer para luego poder hacerlo sin que estas definiciones cambien. La dinámica de los negocios y de la sociedad en si misma hace que las necesidades cambien con frecuencia y hay que estar listos para aceptar estos cambios con naturalidad en nuestros proyectos de software. El mismo cliente irá logrando más claridad en sus ideas a medida que ve crecer el sistema. Con las primeras definiciones inicie su proyecto y mantenga a su cliente cerca e involucrado.

Gestión de la Deuda Técnica.

El concepto de la deuda técnica resulta de mucha utilidad para evaluar las consecuencias futuras de algunas decisiones que a veces se toman sin pensar. Es muy recomendable declarar y analizar cualquier potencial deuda técnica, a todo nivel, desde la arquitectura hasta el código. Mantener la deuda técnica en el plano de la deuda **prudente y advertida**, dará como resultado mejores decisiones y menos sorpresas en el futuro. Ver Capítulo 5.

Detección Temprana de los Defectos.

Poner foco en este tema es fundamental para detectar y solucionar los defectos al poco tiempo de haberlos generado. Fallas habrá siempre, son naturales en esta actividad, el desafío es detectarlas y removerlas lo antes posible. El costo de solucionar una falla aumenta en gran medida conforme avanzan las etapas del proyecto, siendo las peores aquellas fallas conceptuales de las etapas iniciales que llegan a la etapa de producción. El involucramiento del cliente desde las etapas tempranas ayuda mucho en este aspecto. Los métodos ágiles proponen varias prácticas que ayudan también: *Pair Programming*, *Unit Test*, *Test Driven Development*, *Continuous Integration*, etc.

Involucramiento del Cliente en todas las etapas y actividades.

El cliente debe ser parte activa del proyecto, participando, aportando información, compartiendo su visión, validando lo que se hace, y también aprendiendo y dando forma a sus ideas a medida que el proyecto avanza. Si solo se involucra al cliente en la etapa de relevamiento para luego entregar el software varios meses después, seguramente algo habrá cambiado en sus necesidades o algo se habrá hecho mal y será necesario modificarlo. Hay diferentes tipos de clientes, de proyectos y de contratos; seguramente no será posible siempre su presencia permanente, pero es importante buscar en cada caso la mejor forma de maximizar su participación. El contacto fluido con el cliente permite anticipar el cambio en lugar de solo esperar a que los cambios lleguen cuando el trabajo ya está hecho.

Definición y seguimiento de un proceso, actividad disciplinada.

No importa si se usan métodos tradicionales, ágiles, una combinación de ambos, o un proceso propio desarrollado por la empresa. Es importante que el trabajo de las personas esté regulado de alguna manera. Un marco de trabajo debe haber que asegure el mejor uso posible de los recursos y la tecnología por parte de las personas. Bianca Paola Velázquez Rodríguez (2009) menciona en su estudio: "*los individuos y grupos fallan al hacer el mejor uso posible de los recursos disponibles*". Sin un proceso que regule el trabajo, los problemas de comunicación y coordinación terminarán por generar un caos. Incluso los métodos ágiles son partidarios de tener un proceso disciplinado, aunque algunos piensen lo contrario.

Reúso de Componentes, desde la Arquitectura hasta el código.

El reúso parece ser la práctica por excelencia para lograr buena productividad. El secreto está en generar sistemas cada vez más complejos y poderosos haciendo cada vez menos, y esto se puede solo lograr en la medida en que se puedan reusar muchas cosas hechas en proyectos anteriores o que se hayan hecho específicamente para reusar. Una arquitectura basada en componentes con foco en el reúso y la estandarización es clave para obtener piezas de código que luego podrán ser encajadas en otros sistemas. El reúso tiene la gran ventaja de aportar fuertemente a la calidad, ya que todo lo reusado está ya probado y depurado desde sus usos anteriores. Pero para lograr esto, las empresas tienen que ser conscientes e invertir en el desarrollo de patrones de arquitectura reusable y en el desarrollo de los módulos o componentes reutilizables. Nada es gratis. También se pueden comprar componentes hechos en lugar de hacerlos uno mismo, es una opción a evaluar. Como dijo Brooks (1986) en su famoso paper: "*The most radical possible solution for constructing software is not to construct it at all*".

Entrenamiento y Motivación, espacio para decidir, crear, exponer experiencia

Este aspecto es uno de los principales y más valiosos postulados de los métodos ágiles como Scrum, y es también uno de los 7 principios de Lean Software Development. Es importante dar a las personas la oportunidad de usar su conocimiento, poner en juego su experiencia y tomar sus propias decisiones acerca de su trabajo, valorando su capacidad y su profesionalismo. Es importante permitir a las personas exponer sus puntos de vista respecto de cómo enfocar la solución a los problemas, abrir espacios de discusión, de intercambio de opiniones y lograr definiciones conjuntas en los aspectos que hacen al software a desarrollar, su arquitectura, diseño y construcción. Esto generará un ambiente de alta motivación y gran compromiso.

Equipos Chicos y Auto-Gestionados.

Los estudios indican que los equipos grandes suelen padecer de serios problemas de comunicación y coordinación. Respecto del tamaño de los equipos de desarrollo de software, Frederick Brooks dice que "more is less". También Blackburn y Scudder se refieren a esto y lo llaman La Paradoja de la Productividad. Los equipos chicos tienen la

capacidad de trabajar más cohesionados logrando un ambiente de mejor comunicación y mayor colaboración. Si el equipo tiene además la capacidad de organizarse, realizar sus propias estimaciones y asumir sus propios compromisos, se logrará mayor motivación y respuesta a los desafíos.

Timeboxing, tiempos cortos asignados, foco en las entregas frecuentes.

El timeboxing es una forma de referirse a la definición de tiempos cortos para las entregas. Es una buena estrategia para evitar lo que se conoce como el Síndrome del Estudiante, aquello que pasa cuando una persona tiene mucho tiempo para hacer algo, y entonces naturalmente no se preocupa hasta el final cuando ya queda poco tiempo. Es lo que les pasa muchas veces a los estudiantes al preparar un examen con mucha anticipación, de ahí viene su nombre. Timeboxing es una práctica fundamental de los métodos ágiles por la cual el equipo de desarrollo debe estar enfocado en una sola tarea, sin distracciones, por un tiempo corto. Esto asegura máxima productividad. Es la razón de ser de los sprints de 2 a 4 semanas con el objetivo claro de implementar cierta funcionalidad, tal como se los define en Scrum o XP. El foco permanente en una entrega cercana mantiene al equipo motivado y a ritmo altamente productivo, evitando el natural relajo que producen las entregas lejanas.

Lecciones Aprendidas, mejora de los procesos por parte del mismo equipo.

Es fundamental que la experiencia que se obtiene sirva para mejorar el funcionamiento de los equipos de proyecto. Esto permite además que los participantes aporten ideas y sugerencias, aumentando su sensación de pertenencia y su motivación. Lo que se aprende debe ponerse en práctica de inmediato para no volver a caer en los mismos errores y para posibilitar un espiral de mejora, un círculo virtuoso. La experiencia obtenida debe ser compartida, pública, puesta a disposición de toda la organización y la mejora continua debe ser fuertemente apoyada por los niveles más altos del management de la organización.

Alta frecuencia de seguimiento del proyecto (reuniones diarias o cada 2 días).

Es importante para mantener el proyecto alineado con la planificación y las expectativas de los *stakeholders*, que la información sobre el avance y los posibles problemas circule con fluidez para poder tomar acciones rápidas en caso que sea necesario. Los métodos ágiles

llevan al extremo esta práctica con las reuniones diarias del equipo llamadas Daily Meeting donde cada miembro del team debe exponer qué hizo ayer, qué va a hacer hoy, y posibles problemas o demoras. En proyectos donde participan varios equipos de desarrollo es muy importante que se mantengan coordinados con reuniones inter-equipos además de las internas de cada equipo. De nada sirven las reuniones o los informes de avance mensual o quincenal, cuando un problema sea descubierto ya se habrá perdido mucho tiempo. Reuniones frecuentes, cortas, con tiempo limitado, y efectivas (tratando solo temas relevantes) son una buena forma de anticipar y solucionar desvíos y problemas.

Propagación de la información y del conocimiento.

Es fundamental fomentar la comunicación y el trabajo en equipo para que el conocimiento fluya y se propague. Reuniones, presentaciones y espacios de discusión sobre los temas del proyecto deben alcanzar en algún momento a todos los miembros del equipo para asegurar que todos entienden de qué se trata el sistema y como lo que cada uno hace aporta a la solución final. De esta forma cada miembro del equipo podrá aportar mejores ideas y acertará con sus aportes a la solución correcta. Prácticas como la programación por pares con una adecuada rotación de las parejas de programación posibilitan que todo el código sea conocido por al menos 2 o 3 personas eliminando los cuellos de botella que se producen cuando hay algo que modificar o mejorar y solo uno puede hacerlo. Los líderes de equipos juegan un papel fundamental en este aspecto ya que son los que pueden y deben asegurar que la información llegue a los miembros de sus equipos y el conocimiento generado se propague.

Las prácticas de Scrum, XP y TDD

Los métodos ágiles proponen muchas prácticas interesantes que realzan muchos de los aspectos que se han visto en este trabajo como la comunicación, la motivación de las personas, la gestión de los cambios y la detección temprana de los defectos. Aunque no se trabaje con procesos puramente ágiles, es posible tomar ideas e incorporar prácticas como las Daily Meetings, Programación por Pares, Test de Unidad, Test Driven Development, o Integración Continua. Tenga estas prácticas en cuenta y vea cómo puede incorporarlas. Hay mucha literatura al respecto, además de lo explicado en este trabajo.

Collective Code Ownership.

Este aspecto es también muy importante, es un objetivo a alcanzar a partir de lograr un nivel alto de colaboración, comunicación, propagación del conocimiento y trabajo en equipo. CCO se trata de que todos los miembros de un equipo de desarrollo se sientan dueños y responsables por todo el código. El código es de todos, es del equipo, porque todos de alguna u otra manera aportaron para su construcción. Hay que impulsar fuertemente la idea y la cultura de que el código es de todos y todos pueden y deben aportar para mejorarlo y solucionar sus problemas. La práctica de programación por pares a través de la rotación frecuente de las parejas fomenta esto posibilitando que una persona haya trabajado en varias partes del código del sistema, o dicho de otro modo, una porción del código ha sido vista y modificada por varias personas. A la hora de tener que hacer un cambio, habrá varias personas que conocen ese código y que podrán hacerlo porque ya han trabajado con él. Se trata en definitiva de lograr responsabilidad colectiva sobre el resultado final.

Los 7 principios del Lean Software Development.

Y por último, y aunque ya están tratados en los puntos anteriores, no viene mal recordar y tener siempre presentes los 7 principios del Lean Software Development:

#1. Eliminate Waste: todo lo que no aporta valor debe ser eliminado.
#2. Build Quality In: detecte y elimine los defectos poco después de generarlos.
#3. Create Knowledge: el conocimiento debe propagarse, como un virus.
#4. Defer Commitment: decida lo más tarde posible, cuando tenga más información.
#5. Deliver Fast: entregas frecuentes mantienen al cliente involucrado e interesado.
#6. Respect People: dar espacio para pensar, crear y proponer. Eso es también motivar.
#7. Optimise The Whole: mire su proceso, estúdielo. Seguramente pueda mejorarlo.

El Capítulo 2 de este trabajo ha dejado 3 grandes preguntas a responder:

- ¿Cómo es posible medir la cantidad y/o el valor del producto construido, siendo que el software es un intangible?

- ¿Cómo se puede componer un índice de productividad que sirva de base para estudiarla y mejorarla, pudiendo medir el impacto de las acciones que se hagan sobre los procesos, métodos, tecnología, y también sobre las personas?
- ¿Qué se puede hacer para que las horas hombre trabajadas en un proyecto de desarrollo de software rindan al máximo, logrando construir el producto esperado con la menor cantidad de horas posibles?

La tercer pregunta está ya respondida, y aunque no sea posible dar una receta mágica, un método o un proceso que por sí solo lo logre, se han expuesto con claridad una serie de conceptos y factores a tener en cuenta. Queda ahora en la habilidad que cada uno tenga para encontrar la mejor combinación posible de estas buenas prácticas, la que genere el mejor resultado posible en el contexto particular de cada empresa o equipo de desarrollo.

Para medir el tamaño funcional del sistema a construir y luego poder hacer estudios de productividad, es necesario elegir alguno de los métodos disponibles. Use Case Points y Story Points resultan ser los más adecuados y actualizados de los que fueron analizados en el Capítulo 2. Lo importante para esto es seguir el ciclo de la productividad, midiendo, mejorando, y volviendo a medir, siendo disciplinados en la aplicación del método que se use para el cálculo de la productividad. Esto permitirá obtener datos comparables sobre los que será posible analizar la evolución y la efectividad de los cambios que realicemos:

Measure to Know
Know to Change
Change to Lead

La productividad general en actividades de desarrollo de software no está en su nivel óptimo y aún puede ser mejorada. Los conceptos volcados en este trabajo pueden ayudar a lograrlo si se aplican en la forma adecuada en cada empresa o grupo de trabajo. ¡Este es el desafío que nos queda por delante!

Anexo 1 - Bibliografía

JONES, CAPERS: Software Engineering Best Practices, USA, 2010.

JONES, CAPERS: Applied Software Measurement: Global Analysis of Productivity and Quality, USA, 2008.

BELCHER, JOHN G.: Productividad Total (Spanish Edition), 1992.

HERNANDEZ LAOS, ENRIQUE: La productividad multifactorial: concepto, medición y significado, Publicado en "Economía: Teoría y Práctica. Nueva Época", Nro. 26, año 2007.

SUMANTH, D.: Administración para la Productividad Total, Mexico, 1979.

LONGSTREET, DAVID: Function Points Analysis Training Course, Longstreet Consulting Inc.

IFPUG: Function Point Counting Practices Manual, Blendonview Office Park, 5008-28 Pine.

CARROLL, EDWARD: Estimating Software Via Use Cases, USA, 2005.

CLEMMONS, ROY K.: Project Estimation with Use Case Points, USA, 2008.

BROOKS, F.: No Silver Bullet, Essence and Accident in Software Engineering, USA, 1986.

BLACKBURN, J.; SCUDDER, G; WASSENHOVE, L.: Improving Speed and Productivity of Software Development, Vanderbilt University, USA, 1996.

WAGNER S., RUHE M.: A Systematic Review of Productivity Factors in Software Development, Instituto de Informática de la Universidad de Munich, Alemania, 2008.

TYSON R. HENRY: Software Development Productivity: Considering the Socio-Technical Side of Software Development, California State University Chico, 2005.

SCACCHI WALT: Understanding Software Productivity, University of Southern California, Los Angeles, USA, 1994.

VELAZQUEZ RODRIGUEZ P.: Desarrollo de Software, México, 2009.

CURTIS B.: Substantiating Programmer Variability, IEEE (Volume 69, Issue 7), 1981.

BOEHM B.: Improving Software Productivity, TRW Defense Systems Group, USA, 1987.

FOWLER M.: The Technical Debt Quadrant, USA, 2009.

McCONNELL S.: Rapid Development: Taming Wild Software Schedules, USA, 1996.

RATIONAL SOFTWARE Inc.: RUP Best Practices for Software Development Teams, Rational Software White Paper, USA, 2001.

CMU/SEI: CMMI for Development, Version 1.3, USA, 2010.

GIBSON D, GOLDENSON D, KOST K: Performance Results of CMMI Based Process Improvement, USA, 2006.

SCRUM ALLLIANCE: Core Scrum, USA, 2014.

TAHCHIEV P, LEME F, MASSOL V, GREGORY G.: JUnit in Action, USA, 2010.

SCHWABER K., BEEDLE M.: Agile Software Development with Scrum, USA, 2001.

BECK K., ANDRES C.: Extreme Programming Explained: Embrace Change, USA, 2004.

JANZEN D., SAIEDIAN H.: Test-Driven Development: Concepts, Taxonomy, and Future Direction, IEEE Computer Society, 2005.

POPPENDIECK T., POPPENDIECK M.: Lean Sofware Development: An Agile Toolkit, USA, 2003.

AMBLER S., LINES M.: Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise, USA, 2012.

LATTANZE A.: The Architecture Centric Development Method (ACDM), Carnegie Mellon University, USA, 2005.

Anexo 2 – Resultados de la Encuesta

Fue realizada una encuesta on-line en Abril del 2014 que fue respondida por 50 profesionales de la industria de desarrollo de software de Argentina, que se desempeñan en diferentes empresas y funciones. La información y las opiniones volcadas por estos 50 colegas fueron tenidas en cuenta en el desarrollo de este trabajo.

La muestra obtenida es variada por estar integrada por profesionales de diferentes tipos y tamaños de empresas, que ocupan diferentes posiciones, y un alto porcentaje de profesionales con más de 10 años de experiencia en la industria.

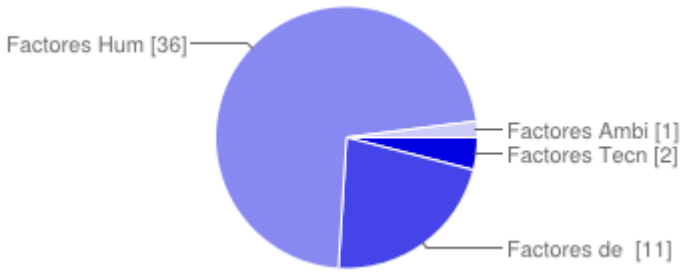
A continuación se vuelca un resumen de los datos obtenidos y algunas conclusiones:

- El 60% de las empresas aproximadamente realiza estimaciones de tamaño funcional, costos y tiempos de los proyectos.
- Los métodos de estimación más usados son Use Case Points, Story Point, Delphi y métodos propios o customizados por las empresas.
- Solo el 50% de las empresas analiza y compara las estimaciones que obtienen al inicio de los proyectos con los resultados obtenidos al finalizar, e intentan determinar las causas de los desvíos.
- Solo en el 30% de las empresas los proyectos finalizan casi siempre cumpliendo las estimaciones, en el 50% de las empresas solo a veces, y en el 20% restante casi nunca.
- El 22% de las empresas utiliza métodos o prácticas formales para el desarrollo en todos los proyectos, y otro 40% solo en algunos proyectos.
- Más del 60% de las empresas utilizan métodos ágiles como SCRUM o XP.
- Solo un 20% de las empresas mide su productividad, y otro 20% lo hace solo a veces o están intentando hacerlo.
- La mayoría de las empresas que calculan productividad lo hacen con valores típicos de la industria como horas hombre, casos de uso, story points, cantidad de retrabajo,

etc. Algunas pocas van más allá y buscan la productividad por valor económico generado (ROI o similar).

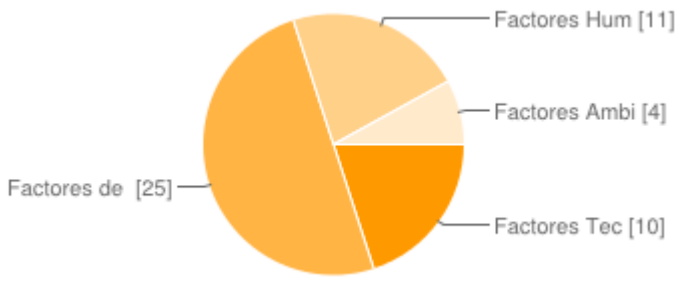
- El mismo 20% aprox de empresas que miden la productividad declaran que realizan acciones para mejorarla y analizan el impacto de las acciones.
- El 50% de las empresas no está conforme con la productividad que obtienen (por medición o por simple percepción), esta opinión coincide en todos los niveles (desde el top management hasta los mismos desarrolladores).
- Sobre los factores, la mayoría de las personan opinan que los factores humanos son los más importantes e influyentes, con especial mención a los temas de capacitación y motivación de las personas y al conocimiento acerca del dominio del software a desarrollar y la calidad de las especificaciones.
- Factores humanos y de procesos los que más influyen en la productividad para la gran mayoría de los encuestados:

Si agrupamos los factores que pueden aumentar o reducir la productividad en desarrollo de software en los 4 grupos abajo listados; ¿cuál te parece que es el grupo más relevante?



Factores Tecnológicos (equipos, herramientas).	**2**
Factores de Procesos (metodologías, estándares, etc.).	**11**
Factores Humanos (entrenamiento, motivación, etc.).	**36**
Factores Ambientales (tipo de oficinas, espacio, luz, ruido, etc.).	**1**

¿Y cuál te parece que es el segundo grupo en relevancia?



Factores Tecnológicos (equipos, herramientas).	**10**
Factores de Procesos (metodologías, estándares, etc.).	**25**
Factores Humanos (entrenamiento, motivación, etc.).	**11**
Factores Ambientales (tipo de oficinas, espacio, luz, ruido, etc.).	**4**

Anexo 3 – Transcripción de Entrevistas

Fue realizada una entrevista al fundador y Presidente de una importante empresa argentina dedicada al desarrollo de software, consultoría en temas de ingeniería de software, mejora de procesos y otros temas relacionados. La persona entrevistada fue seleccionada por ser uno de los grandes referentes de la actividad en Argentina, con muchos años de experiencia nacional e internacional. Se describen a continuación las preguntas que guiaron la entrevista, las conclusiones y aspectos relevantes que aportaron a este trabajo.

Preguntas Utilizadas como Guía de la Entrevista

- ¿Es posible obtener mediciones objetivas de productividad en desarrollo de software? ¿Cómo? ¿Con qué información?
- ¿Cómo medís la cantidad de producto construido para ponerlo como divisor de la fórmula de productividad?
- ¿Qué es lo que hace que una organización sea más productiva que otra? ¿Cuáles son los factores principales?
- ¿Consideras que algún factor tiene una importancia superior o especial por sobre los otros?
- En el segmento de empresas que comentamos, las PYMES Argentinas, ¿ves que esto está bien trabajado? ¿Hay cosas para hacer? ¿Hay para aportar?
- ¿Qué Datos de productividad se publican en otros países?
- ¿Se usan medidas como Puntos de Función o Use Case Points?
- ¿Te parece aplicable hoy en el desarrollo moderno, estos métodos que ya tienen muchos años? ¿Son aplicables hoy y vigentes? ¿O te parece que hay algo que viene que se está perfilando para reemplazarlos?
- Encuentro en ACDM una forma de avanzar en sprints de desarrollo ágiles sobre la base de una arquitectura que responde las grandes preguntas técnicas. ¿Qué te parece?

Aspectos Relevantes y Conclusiones

- Todo esto tiene mucho que ver con la materia prima con la cual se opera. Producir software es transformar conocimiento. Es complejo, pero es posible medir la productividad.
- Los tres grandes factores: el proceso, la gente y la tecnología. Lo que importa es cómo estos factores se combinan.
- La relación entre la productividad y la calidad, el programador rápido (con retrabajo) y el más lento con mejor calidad. ¿Cuál es el mejor?
- ¿Cómo hago para mejorar productividad, sin perder calidad? Reúso es la práctica fundamental, hacer cada vez menos. El reúso es un factor de proceso.
- ¿Qué hace que un individuo dado un proceso, tenga buena productividad? Capacitación, Motivación, Skills, etc. Hay mucha gente va al trabajo y no a trabajar.
- La importancia de la organización del tiempo y del trabajo, reuniones productivas, que no haya tiempos muertos, disponibilidad de recursos para que el trabajo no se frene, forma de asignar el trabajo (agrupado en proyectos y no asignaciones sueltas), evitar el síndrome del 90%, evitar el síndrome del estudiante.
- El otro componente es el tecnológico, se produce mucho "overhead" si la tecnología no es la adecuada. La tecnología debe facilitar las transiciones por ejemplo.
- El concepto de productividad son estas tres cosas alineadas: Procesos, Personas y Tecnología.
- El factor más importante es el humano… El resto de los factores acompañan y son complementarios. El tema de la gente es fundamental.
- En Argentina no hay datos. La industria Argentina no tiene datos acerca de cuál es la productividad, no existen datos como presentación de país que pretende exportar software.
- Una de las cosas que complica mucho los estudios de productividad es que nunca nos asentamos sobre un método sobre el cual trabajar. La ingeniería de software tiene esa cuestión adolescente de que como esto no me gusta busco otra cosa aunque todavía no hice el esfuerzo por entender bien qué significa.

- Crunch Mode y Rapid Application Development, antecedentes poco probados de las metodologías ágiles.
- El cuarto factor: el dominio, la complejidad de lo que se hace. No puedo comparar la productividad de alguien que trabaja en sistemas de misión crítica, con alguien que trabaja en otro tipo de software.
- Deuda Técnica, postergar decisiones técnicas no visibles para el usuario pero que tienen un impacto muy fuerte en la forma en la que el sistema va a funcionar y operar. Los métodos ágiles generan deuda técnica para entregar valor rápido y luego viene el refactoring. Pueden perder el beneficio de entregar rápido. Martin Fowler tiene una muy buena definición de qué es deuda técnica.
- Más aceptado hoy en los métodos ágiles que los primeros sprints son para definiciones técnicas, arquitectura, etc., para reducir la deuda técnica.
- El tradeoff que hay entre Calidad y Velocidad (Time to Market), afecta mucho las decisiones técnicas y en definitiva la productividad.
- Hay 3 cosas importantes:
    - Gestionar y Compartir el conocimiento del grupo.
    - La automatización del proceso.
    - Los procesos colaborativos con stakeholders integrados (DevOps).
- Es fundamental saber entender la diferencia que hay entre agilidad (adaptarse a los cambios) y velocidad (hacer menos, reusar más). Libro de Caper Jones, Best Practices for Software Development, pone en el tope las mejores prácticas al reúso.
- El concepto de Product Line aporta ideas interesantes para mejorar la productividad.
- Para reusar hay que invertir, nada es gratis. La organización tiene que estar consciente de esto.
- Generar modelos de referencia de arquitectura que ya contengan las grandes decisiones técnicas, luego usarlos de input para desarrollos ágiles. Pero esto implica invertir.
- El concepto de productividad pasa también por el reúso, no solo la cantidad de software que se produce, sino también lo que no se produce por el reúso.
- La importancia de cómo la organización ve esto para poner foco, sino siempre va a ganar la velocidad por sobre la productividad.