UNIVERSIDAD NACIONAL DE CATAMARCA
FACULTAD DE TECNOLOGÍA Y
CIENCIAS APLICADAS

LICENCIATURA EN SISTEMAS DE INFORMACION

TRABAJO FINAL

LEAN SOFTWARE DEVELOPMENT
APLICACIÓN:
SISTEMA DE GESTIÓN TÉCNICA DE FIBRA
COTECA S.A.

Autores:

PIZA, CARLOS SEBASTIAN **MU Nº 2054**

MASCAREÑO, MARIA CECILIA **MU Nº 2028**

Profesor Guía:

LIC. FABBRIS, DOMINGO ARIEL

MARZO 2018

1. INDICE

2. RESUMEN

Toda organización dedicada al desarrollo de software utiliza un proceso para la creación de sus productos. En la mayoría de los casos el proceso se adapta al tamaño, recursos y necesidades de cada organización, pero las características básicas se basan en una (o más) de las metodologías de desarrollo "estándar". Podríamos decir que el término se refiere al marco que se utiliza para estructurar, planificar y controlar el proceso de desarrollo de un sistema de información.

El presente trabajo trata sobre la metodología ágil de desarrollo de software denominada Lean Software Development. Se presentarán las pautas y principios que dan base a la misma. Se detallarán luego las distintas prácticas asociadas a la implementación de la metodología y luego, a modo de caso de estudio, se realizó un sistema para la gestión técnica de la materia prima de la empresa COTECA S.A., en donde se pusieron en uso los conceptos previamente desarrollados. Finalmente se muestran los resultados obtenidos en función del valor obtenido al hacer uso de la filosofía Lean.

La investigación fue dividida en dos grandes partes. La primera fase fue una exploración de la teoría de Lean y sus principios de pensamiento. Esto se hizo mediante la lectura de la literatura existente sobre el tema. Eso ayudó a obtener una comprensión más profunda de los supuestos teóricos de Lean y de sus prácticas y principios. En esto, se puso interés primordial en tres conceptos clave de la filosofía: *costos*, *calidad* y *respeto por las personas*. El conocimiento obtenido en esta fase formó los cimientos teóricos para el desarrollo y análisis de la segunda fase.

La segunda fue desarrollar el sistema que cumplió el rol de caso de estudio, comparando los resultados de la implementación de Lean con los resultados teóricos de la primera etapa a fin de validarlos.

El objetivo principal de este trabajo es mostrar el potencial que posee esta metodología para el desarrollo de aplicaciones, independientemente del ámbito de implementación. Este potencial se basa en la propuesta de técnicas sencillas de mejora de rendimiento y calidad de los recursos aplicados, lo que permitirá detectar e identificar fuentes de perfeccionamiento en el trabajo mediante la eliminación o reducción de lo que en la metodología se denomina *desperdicio*.

Se demostrará mediante la aplicación de los principios y prácticas que dan sustento al desarrollo de software *Lean*, que se puede obtener un producto lo más cercano posible a lo requerido por el usuario y de una manera mucho más "magra", centrándose únicamente en lo que aporta valor real desde el punto de vista del cliente.

3. INTRODUCCION

El desarrollo de software ha estado plagado de tasas de éxito sorprendentemente bajas por décadas. Al mismo tiempo, el número de productos y servicios controlados por software continúa incrementando considerablemente cada año. Si estas fueran las dos únicas tendencias, es evidente que nos encaminaríamos hacia un desastre.
Afortunadamente, los métodos ágiles de desarrollo de software han manifestado que el aumento de las tasas de éxito son posibles y las técnicas Lean (que han demostrado su eficacia por más de 50 años en las industrias de fabricación) ahora están siendo aplicadas al desarrollo de software y validando los éxitos del desarrollo ágil.
Las metodologías ágiles fueron los primeros intentos de mejorar el desarrollo de software, entrando Lean en escena mucho tiempo después.
En la década de 1990 hubo una creciente insatisfacción con el software creado con metodologías y procesos de desarrollo tradicional. El uso de estos procesos no resolvió ninguno de los problemas habituales del desarrollo de software: alta tasa de fracasos del proyecto, baja calidad del software y clientes generalmente disconformes. Esto dio lugar a una serie de metodologías alternativas, incluyendo Extreme Programming (XP), Scrum, Método de Desarrollo de Sistemas Dinámicos (DSDM), Crystal, entre otros, que se conocen colectivamente como métodos livianos. Este término fue introducido para diferenciarse de los métodos tradicionales predominantes de la época. Sus creadores no estaban contentos con el término “liviano”, ya que daba a entender que estos métodos eran menos amplios o menos importantes. Así, en febrero de 2001, se realiza la famosa reunión Snowbird, en donde se decide utilizar el término “Ágil” por considerarlo más representativo.
Todas las metodologías ágiles comparten una serie de prácticas o características básicas que las diferencian de un proceso tradicional. Estas resumen el espíritu ágil y determinan el proceso a seguir por el equipo de desarrollo en cuanto a metas y organización del mismo. Estas prácticas pueden resumirse en:

- La prioridad es satisfacer al cliente mediante tempranas y continuas entregas de software que le aporte un valor.
- Dar la bienvenida a los cambios. Se capturan los cambios para que el cliente tenga una ventaja competitiva.
- Entregar frecuentemente software que funcione desde un par de semanas a un par de meses, con el menor intervalo de tiempo posible entre entregas.
- La gente del negocio y los desarrolladores deben trabajar juntos a lo largo del proyecto.
- Construir el proyecto en torno a individuos motivados. Darles el entorno y el apoyo que necesitan y confiar en ellos para conseguir finalizar el trabajo.
- El dialogo cara a cara es el método más eficiente y efectivo para comunicar información dentro de un equipo de desarrollo.
- El software que funciona es la medida principal de progreso.
- Los procesos ágiles promueven un desarrollo sostenible. Los promotores, desarrolladores y usuarios deberían ser capaces de mantener una paz constante.
- La atención continua a la calidad técnica y al buen diseño mejora la agilidad.
- La simplicidad es esencial.

- Las mejores arquitecturas, requisitos y diseños surgen de los equipos organizados por sí mismos.
- En intervalos regulares, el equipo reflexiona respecto a cómo llegar a ser más efectivo, y según esto ajusta su comportamiento.

Estas prácticas son compatibles con el desarrollo de software Lean, que es el objeto de investigación de este trabajo.
La aplicación de los principios Lean al desarrollo de software es algo reciente. En un principio comenzó como el proceso de fabricación Lean, aproximadamente entre 40 y 60 años atrás, llamándoselo en ese momento Sistema de Producción Toyota o fabricación Just-In-Time. James Womack, Daniel Jones y Daniel Roos acuñaron el término “Lean” (magro) en su libro de 1990, “The Machine ThatChangedtheWorld”.
Lean es una mentalidad, una manera de pensar acerca de cómo entregar valor a los clientes con mayor rapidez mediante la búsqueda y eliminación de residuos (que son los impedimentos principales para lograr calidad y productividad).
Los principios Lean y la mentalidad del pensamiento Lean han probado ser muy aplicables para la mejora de la productividad y la calidad de casi cualquier empresa. Lean ha sido aplicado con éxito en la fabricación, distribución, cadena de suministros y desarrollo de productos, ingeniería, bancos y muchos ámbitos más. Sin embargo, solo en los últimos años los principios y técnicas Lean se han aplicado para el desarrollo de software.
En 2003, Mary y Tom Poppendieck publicaron el primer mapa completo de los principios Lean para el desarrollo de software. Luego en 2006 refinan este mapeo y publican su libro “Implementing Lean Software Development: From Concept to Cash”.
La mayoría de los escritos posteriores sobre esta metodología han seguido los lineamientos introducidos por los Poppendieck, utilizando los siete principios que ellos definieron:

- Eliminar desperdicio
- Ampliar el aprendizaje
- Decidir lo más tarde posible
- Entregar lo más rápido posible
- Potenciar al equipo
- Crear integridad
- Ver todo el conjunto

A lo largo del presente trabajo abordaremos en detalle cada uno de los mismos, junto con las distintas herramientas y prácticas necesarias para su aplicación, brindando ejemplos concretos que podrán ser valorados en función de sus resultados y apego a los principios de la filosofía que se está presentando.

4. MARCO TEORICO

4.1. INGENIERIA DE SOFTWARE.

4.1.1. EL PROBLEMA CON EL DESARROLLO DE SOFTWARE

Al emprender cualquier proyecto de desarrollo de software, normalmente se corre el riesgo de tener alguno (o en peor de los casos, todos) los siguientes problemas:

- Sobrepasar los tiempos del cronograma planificado.
- Generar costos por encima del presupuesto fijado.
- No cumplir con las necesidades del cliente.
- Cancelación del proyecto.

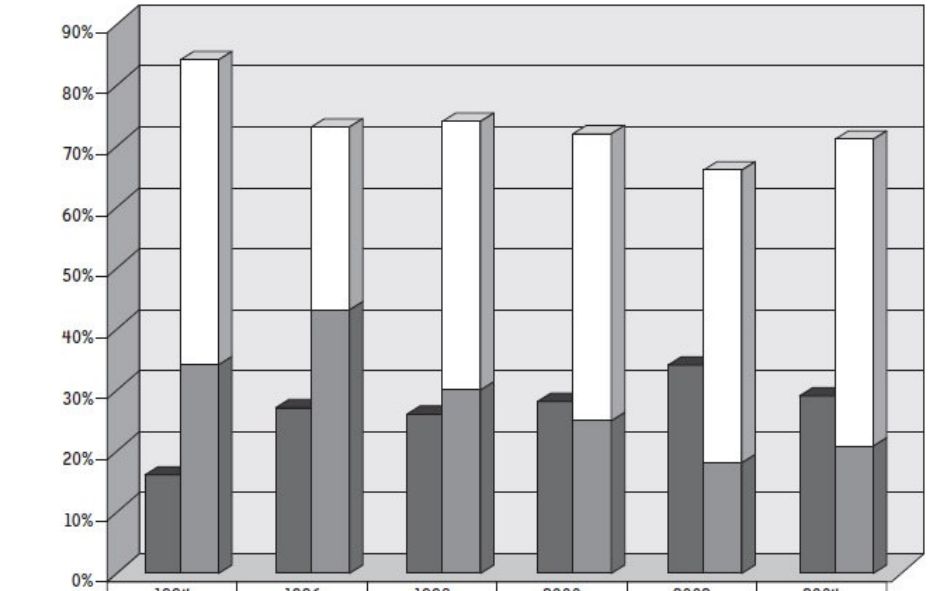
El informe del Grupo Standish de 1994 llamado *CAOS* fue el estudio fundamental sobre el fracaso de los proyectos de tecnologías de información (IT). En ese año se estudiaron más de 8000 proyectos de desarrollo de software y se encontró que solo el 16% fueron exitosos. Esto significa que 84% de los proyectos habían fallado directamente o tenido serios problemas. En 2004, después de 10 años de estudio, el número de proyectos incluidos había aumentado a 40.000 y la tasa de éxito había trepado al 29%, lo que fue un aumento significativo pero para nada extraordinario. De hecho, se hace difícil encontrar otro ejemplo de una industria con una tasa de éxito tan baja.

4.1.2. EL ESTUDIO CAOS

El estudio del Grupo Standish incluyó grandes, medianas y pequeñas empresas en los principales segmentos de la industria: bancario, seguridad, manufactura, ventas por menor y mayor, salud, seguros, servicios y organizaciones estatales (365 empresas en total). Durante un periodo de 10 años se estudió más de 40.000 proyectos, utilizando grupos de debate y entrevistas personales para proporcionar un contexto cualitativo para los resultados de la investigación. Se llegó a una clasificación de los proyectos dividida en tres categorías:

Proyecto Exitoso: se completó a tiempo y dentro del presupuesto con todas las características y funciones que se especificaron originalmente.
Proyecto Desafiante: fue completado, pero superó el presupuesto, el tiempo estimado y ofreció un menor número de características y funciones que las planeadas en un principio.
Proyecto Fallido: fue cancelado en algún punto durante el ciclo de desarrollo.

En la siguiente figura se muestra el porcentaje de proyectos en cada una de esas categorías durante un periodo de 10 años.



	1994	1996	1998	2000	2002	2004
Exitoso	16%	27%	26%	28%	34%	29%
Desafiante	53%	33%	46%	49%	51%	53%
Fallido	31%	40%	28%	23%	15%	18%

Figura 4-1. Datos del estudio *CAOS*

Es justo mencionar que algunos expertos no están de acuerdo con la definición de éxito usada en estudio. Señalan que muchos proyectos que se consideraron “desafiantes” se transformaron en productos de gran éxito. Sin embargo, incluso tomando esto en cuenta, la tasa de éxito dejaba mucho margen para la mejora.

4.1.3. EL MÉTODO DE CASCADA

Si nos preguntamos a que se debe una tasa de fracaso tan alta, una gran parte de culpa puede atribuirse a la adopción generalizada del Método de Cascada.
El modelo de desarrollo de software en cascada divide el proceso en una serie de fases distintas que se realizan de forma secuencial: requerimientos, diseño, implementación, pruebas, despliegue y mantenimiento. El desarrollo es visto como si fluyera constantemente hacia abajo (al igual que una cascada) a través de las fases. Cada una de éstas tiene un principio y un fin, y una vez que se pasa a la siguiente fase no se puede volver atrás.

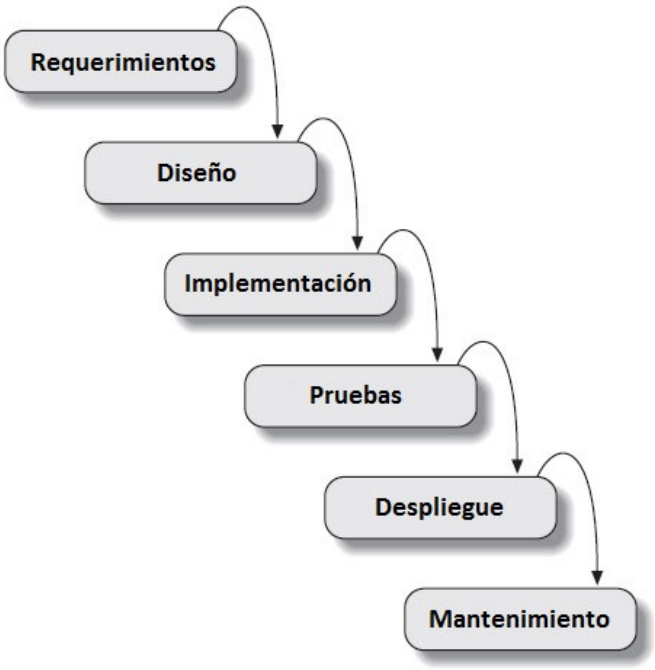


Figura 4-2. Modelo de Cascada

El método de cascada tiene un atractivo aire de simplicidad. Los gerentes gustan tener una serie de hitos fijos que hagan que seguir el progreso de un proyecto parezca fácil. Sin embargo, esta flexibilidad es una ilusión porque el modelo no permite cambios.
El desarrollo de software es un proceso altamente dinámico y el cambio es inevitable. Durante la implementación es muy probable que se encuentren problemas de diseño, ya que los clientes no saben exactamente lo que quieren en un principio y eso llevará a que se presenten cambios en los requisitos.
Estudios demuestran que el método de desarrollo en cascada se correlaciona positivamente con la productividad reducida, aumento de las tasas de defectos y fracaso del proyecto. Inevitablemente se plantea la interrogante de cómo este método llegó a ser tan promovido y a estar tan arraigado pese a tanta evidencia en su contra.
Winston Royce describió por primera vez el método de cascada en un artículo de 1970 titulado “Gestión del Desarrollo de Grandes Sistemas de Software”. En este escrito se cita a menudo como si validara el modelo de cascada, pero en realidad hace lo contrario. Las personas que utilizan el trabajo de Royce para apoyar este método probablemente no lo hayan leído con cuidado, ya que dice explícitamente que el método de cascada “es riesgoso e invita al fracaso” y recomienda la búsqueda de un estilo de desarrollo iterativo.
El método de cascada probablemente habría ido desapareciendo poco a poco, pero en la década de 1980 se convirtió en el estándar para desarrollo y adquisición de software del Departamento de Defensa de Estados Unidos (DoD) con el lanzamiento del DOD-STD-2167. Con el tiempo, el DoD se dio cuenta de que el método adoptado no estaba funcionando y en 1994 fue reemplazado con el MIL-STD-498, que apoya el desarrollo iterativo. Sin embargo, el daño ya estaba hecho y una fuerte mentalidad con un sesgo hacia el desarrollo en cascada se había arraigado. Los métodos “livianos” de la década de 1990 y los ágiles de la década del 2000 cambiaron esto.

4.1.4. METODOLOGIAS AGILES

Existe un gran número de metodologías formales que caen dentro del territorio Ágil. Aunque varían en sus actividades y artefactos específicos, todas ellas utilizan iteraciones cortas de tiempo definido que entregan software funcional al final de cada una de ellas (la longitud de las iteraciones varía entre las metodologías)

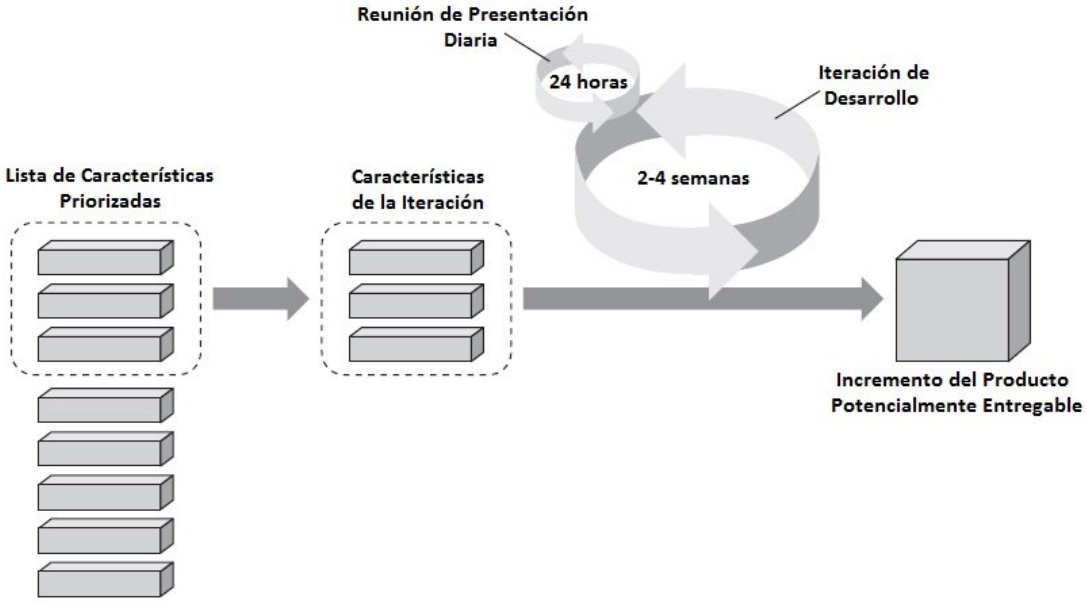


Figura 4-3. Típico Proceso Ágil

Algunas de las metodologías ágiles comúnmente más usadas son:

Scrum: tomó con éxito el enfoque ágil y lo redujo a lo esencial. El resultado es una de las metodologías ágiles más simples y fáciles de implementar que todavía provee el desarrollo de software ágil.
XP: tiene un nutrido conjunto de prácticas que pueden ser abrumadoras para los principiantes en el camino ágil, pero se lleva el crédito por popularizar la mayor parte de las prácticas fundamentales adoptadas por otras metodologías.
Crystal: es en realidad una familia de metodologías creada por AlistairCockburn. Los procesos y las prácticas varían dependiendo del tamaño, criticidad y complejidad del proyecto.
FDD: su perspectiva se centra en la creación de modelos de dominio para el sistema que se está creando, que luego organiza el desarrollo en torno a las características que el modelo implementa. Esto hace a FDD única entre las metodologías ágiles.
Proceso Unificado (UP): considerado generalmente uno de los procesos ágiles más "pesados" a pesar de estar destinado a ser adaptado. De todos modos, esto ha llevado a una serie de variantes, incluyendo Proceso Unificado Racional (RUP), el Proceso Unificado Ágil (AUP) y el Proceso Unificado Empresarial (EUP).

DSDM: es más formal que la mayoría de los métodos ágiles, especificando totalmente los diferentes roles, procesos y objetos. Una característica notable es la priorización de los requisitos.

Todos estos métodos nacieron en diferentes momentos de la década de 1990 como respuesta al fracaso del método de cascada. Ha habido un gran enriquecimiento mutuo de ideas y técnicas entre estos métodos.

5. MELODOLOGÍA ÁGIL DE DESARROLLO LEAN

5.1.1. PRIMER PRINCIPIO - ELIMINAR DESPERDICIO

Para realmente apreciar el surgimiento de la producción Lean y sus derivados, se tiene que entender con qué se estaba compitiendo y qué se quería reemplazar: **la producción en masa**.

Henry Ford popularizó la producción en masa mediante la fabricación con línea de montaje del Modelo T en 1913 (que a su vez había sustituido a la producción artesanal). La producción en masa se utiliza para fabricar grandes cantidades de productos a un costo unitario bajo. Se divide el proceso de fabricación en pequeños pasos que se pueden llevar a cabo por mano de obra no especializada, basándose en el uso de maquinaria de alta precisión y partes estandarizadas intercambiables.

El inconveniente de la producción en masa es su inflexibilidad. Una línea de montaje de producción en masa es cara de instalar y difícil de alterar, el proceso solo es económico si se va a producir grandes cantidades de la misma cosa.

A fines de la década de 1940, la empresa Toyota se estableció en Japón para la fabricación de automóviles, pero existía un problema: dado que las personas no tenían dinero, los coches debían ser baratos. La producción en masa fue la forma más barata de hacer automóviles, pero significaba fabricar miles de unidades del mismo tipo, y el mercado japonés no era lo suficientemente grande como para justificar estas cantidades. Entonces el dilema que afrontaba Toyota era fabricar automóviles en pequeñas cantidades pero producirlos en serie de la manera más barata posible.

TaiichiOhno se dio cuenta que el sistema americano de producción en masa no funcionaría en Japón. El mercado interno de automóviles era demasiado reducido y había una gran demanda de variedad en los productos, desde pequeños y económicos a grandes y lujosos. Por necesidad, TaiichiOhno experimento con muchas ideas y técnicas que eventualmente se convirtieron en el Sistema de Producción Toyota, que él mismo describió como “un sistema para la eliminación absoluta de residuos”.

Residuo parece un término razonablemente claro, pero Ohno dio un nuevo significado a esa palabra. En su razonamiento, todo lo que no crea valor para el cliente es un residuo. Una pieza que está ociosa esperando a ser utilizada, hacer algo que no se necesita de inmediato, movimientos innecesarios, transporte, tiempos de espera, pasos adicionales de procesamiento y, por supuesto los defectos, son RESIDUOS.

Ohno no trataba de imitar la producción en masa, por lo que no adopto los principios de ese estilo de manufactura. Su objetivo era tanto hacer y entregar un producto inmediatamente después que un cliente efectuara un pedido. El creía que era mejor esperar la llegada de un pedido de un cliente en vez de generar stock anticipando la llegada de un posible pedido.

Toyota trasladó su concepto de RESIDUO o DESPERDICIO desde el proceso de fabricación al desarrollo de productos. Cuando se inicia un proyecto de desarrollo, el objetivo es terminar lo más rápido posible, porque todo el trabajo invertido en el desarrollo no suma valor hasta que el automóvil no sale de la línea de producción. En cierto sentido, los proyectos de desarrollo en curso pueden considerarse como stock ocioso alrededor de la

fábrica. Para los clientes, los diseños y prototipos no son útiles; ellos reciben valor cuando el nuevo producto es entregado.
Parece extraño considerar desperdicio a los pasos intermedios de un programa de desarrollo, pero en la década de 1980 parecía igualmente extraño que el stock debiera considerarse desperdicio. Después de todo, mantener un determinado nivel de stock fue lo que permitió realizar la entrega inmediata al cliente una vez realizado el pedido y posibilitó que la maquinaria productiva funcionara a su máxima capacidad.
En realidad, la acumulación de stock es una gran pérdida. Las maquinas funcionando a su máxima capacidad productiva generan grandes cantidades de stock innecesario que ocultan problemas de calidad, se vuelve obsoleto y obstruye los canales de distribución. Un proceso de desarrollo largo o con retrasos adolece de los mismos inconvenientes.
La eliminación de desperdicios es el principio fundamental del desarrollo Lean, del cual surgen los demás principios. Por lo tanto, el primer paso para la implementación del desarrollo Lean es aprender a identificar los desperdicios. El segundo paso es descubrir las mayores fuentes de residuos y eliminarlas. El siguiente paso es identificar las fuentes restantes y eliminarlas e ir repitiendo este proceso. Después de un tiempo, incluso las cosas que parecen esenciales pueden ser eliminadas gradualmente.
Muchas organizaciones han intentado aplicar el pensamiento Lean pero no han tenido un éxito uniforme en ello, ya que se requiere un cambio en la cultura organizacional y hábitos que está más allá de las capacidades de algunas empresas. Por otro lado, las empresas que han entendido y adoptado la esencia del pensamiento Lean han obtenido mejoras de rendimiento significativas y sostenibles.
Los principios son ideas guía y puntos de vista acerca de la disciplina. Son universales, pero no siempre es fácil ver cómo aplicarlos en entornos específicos. Para ello se utilizan las denominadas PRÁCTICAS, que son las que realmente permiten llevar a cabo los principios. Dan orientaciones específicas sobre qué hacer, pero deben ser adaptadas al entorno de aplicación. Podría decirse que no existen prácticas mejores que otras; deben tener en cuenta el contexto. De hecho, los problemas que surgen en la aplicación de metáforas de otras disciplinas al desarrollo de software a menudo son el resultado de intentar transferir las prácticas en lugar de los principios de la otra disciplina.
El desarrollo de software es una disciplina amplia que puede abarcar desde el diseño web hasta el envío de un satélite en órbita. Las prácticas para un dominio no necesariamente se aplicarán a otros dominios. Sin embargo los principios son de aplicación general en todos los dominios, siempre y cuando los principios guía se traduzcan en prácticas adecuadas para cada dominio.
Existen dos requisitos previos para afianzar una nueva idea en una organización:

Se debe demostrar que la idea funciona operativamente.
Las personas que están considerando adoptar el cambio deben entender por qué éste funciona.

Las prácticas ágiles de desarrollo de software han demostrado funcionar en algunas organizaciones, y el desarrollo Lean amplía los fundamentos teóricos del desarrollo ágil mediante la aplicación de los conocidos y aceptados principios Lean llevados al desarrollo de software.

5.1.1.1. IDENTIFICAR EL DESPERDICIO

Aprender a identificar los residuos es el primer paso en el desarrollo de innovaciones con el pensamiento Lean. Si hay algo que no aporta directamente valor según la necesidad del cliente, se trata como desperdicio. Si existe una manera de hacer las cosas prescindiendo de algo, ese algo es desperdicio. En 1970, Winston Royce escribió que los pasos fundamentales del desarrollo de software son el análisis y la codificación, pero también se requieren muchos pasos adicionales que no contribuyen tan directamente en el producto final como lo hacen el análisis y la codificación, generando un aumento en el costo de desarrollo. Con la definición de residuo empleada en la metodología, se puede interpretar el estudio de Royce para indicar que todos los pasos del modelo tradicional de desarrollo en cascada, excepto el análisis y la codificación, son desperdicios.[1]
Las prácticas del desarrollo ágil de software buscan eliminar los residuos, y para ello, primero es necesario identificarlos. Un buen punto para comenzar sería buscar todo lo que hace una organización para desarrollar software que no sea análisis o codificación.
ShigeoShingo, uno de los autores intelectuales del Sistema de Producción Toyota, identificó siete tipos de residuos en el proceso de fabricación.[2] Esta clasificación ha ayudado a que en el proceso de manufactura se encuentren residuos en donde nunca se había pensado buscar. En el proceso de desarrollo de software, la clasificación se ha adaptado para ayudar a la búsqueda de residuos, como puede verse en la tabla 5.1

Los Siete Desperdicios

En la Manufactura	En el Desarrollo de Software
Stock Excesivo	Trabajo Parcialmente Hecho
Procesamiento Adicional	Procesos Adicionales
Exceso de Producción	Características Adicionales
Transporte	Conmutación de Tareas
Esperas	Esperas
Movimiento	Movimiento
Defectos	Defectos

Tabla 5.1

5.1.1.2. Trabajo Parcialmente Hecho

En el desarrollo de software, el trabajo parcialmente hecho tiene la tendencia a convertirse en obsoleto e interfiere en otros desarrollos que podrían necesitar ser realizados. Pero el gran problema con el software parcialmente hecho es que no se tiene idea de que si al finalizarlo funcionará o no. Se puede tener una gran cantidad de requisitos y documentos para el diseño, incluso se puede tener código o hasta la unidad de prueba; pero mientras todo esto no se integre con el resto del entorno, no se sabrá muy bien que problemas podrían aparecer, y hasta que el software no esté realmente en producción, no se sabe si resolverá el problema que justificó su desarrollo.
El desarrollo parcialmente hecho ata recursos en inversiones que aún no han dado resultados. Estas inversiones son a veces capitalizadas y la depreciación se inicia cuando el software comienza a funcionar. Si el software nunca entra en producción, entonces podría considerarse que la inversión se perdió. El desarrollo de software parcialmente hecho puede

conducir a enormes riesgos financieros. Llevar este factor al mínimo es una reducción del riesgo, como así también una estrategia de reducción de desperdicios.

5.1.1.3. Procesos Adicionales

Los trámites administrativos consumen recursos, incrementan el tiempo de respuesta, esconden problemas de calidad, se extravían, se degradan y se convierten en obsoletos. Los trámites que nadie se preocupa de leer no agregan valor.
Muchos procesos de desarrollo de software requieren documentación para cerrar tratos con el cliente, para proporcionar seguimiento o para obtener la aprobación de un cambio. Pero el hecho de que el papeleo sea algo necesario de entregar, no significa que agregue valor al cliente. Si se debe producir documentación que aporte poco valor al producto, hay tres reglas a tener en cuenta:

- Que la documentación sea lo más breve posible.
- Que sea de alto nivel.
- Hacerlo fuera de línea.

Los sistemas de seguridad crítica con frecuencia se regulan, y para ello se necesitan requerimientos por escrito que puedan ser remontados al código. En este caso, la administración de los requisitos en un formato específico para que puedan ser fácilmente evaluados y verificados por completo, se puede calificar como una actividad que genera valor añadido.
Una buena prueba de valor de la documentación es ver si alguien está esperando por lo que se produce. Si un analista completa planillas, hace tablas o escribe casos de usos que otros están dispuestos a utilizar para codificar, probar y escribir manuales de instrucciones, entonces esta documentación agrega valor. Sin embargo, debe haber una constante búsqueda de medios más eficaces y eficientes para transmitir la información.

5.1.1.4. Características Adicionales

Puede parecer una buena idea agregar algunas características adicionales a un sistema en caso de que sea necesario. Los desarrolladores tal vez deseen añadir una nueva capacidad técnica para ver cómo funciona. Esto puede ser inofensivo, pero por el contrario, se trata de un serio residuo. Cada fragmento de código en el sistema tiene que ser seguido, compilado, integrado y probado cada vez que se modifica, y por ende, tiene que ser mantenido durante toda la vida del sistema. Cada línea de código aumenta la complejidad y es un punto potencial de errores. Hay una gran posibilidad de que el código adicional se convierta en obsoleto antes de que sea utilizado, ya que después de todo no existía una necesidad real para éste. Si el código no es necesario ahora, su adición al sistema es un desperdicio.

5.1.1.5. Conmutación de Tareas

Asignar personas a varios proyectos es una fuente de desperdicio. Cada vez que un desarrollador cambia entre una tarea y otra, incurre en un tiempo de cambio significativo hasta que se logra enfocarse en la nueva tarea.[3] Pertenecer a múltiples equipos de trabajo por lo general causa más interrupciones, y por lo tanto, más cambios de tarea, siendo esto

un desperdicio. La forma más rápida de terminar dos proyectos que utilizan los mismos recursos es hacer uno a la vez.
Puede resultar tentadora la posibilidad de iniciar varios proyectos simultáneos, pero el exceso de trabajo en una organización dedicada al desarrollo de software crea una gran cantidad de residuos, ya que en realidad se retrasa el trabajo.

5.1.1.6. Esperas

Uno de los mayores residuos en el desarrollo de software usualmente es la espera de que las cosas sucedan. Los retrasos en el inicio de un proyecto, en la dotación de personal, retrasos debidos a excesiva documentación de requisitos, demoras en las revisiones y aprobaciones, retrasos en las pruebas y en la implementación son residuos. Los retrasos son comunes en la mayoría de los procesos de desarrollo de software, y parecería contradictorio pensar en estos como un residuo, pero la velocidad de respuesta a una necesidad crítica y urgente del cliente está directamente relacionada con los retrasos sistemáticos en el ciclo de desarrollo en la organización.
Uno de los principios fundamentales del desarrollo Lean es posponer las decisiones hasta el último momento posible, de manera tal de poder tomarlas con la mayor cantidad de información posible. Este es un enfoque basado en opciones para desarrollar software, y es la mejor forma de lidiar con la incertidumbre.

5.1.1.7. Movimiento

El desarrollo de software es una actividad que requiere gran concentración, por lo que trasladarse de un lugar a otro genera más retrasos de lo que se piensa. Cuando un desarrollador tiene una pregunta y debe buscar la respuesta lejos de su puesto de trabajo, probablemente le tomará más tiempo restablecer la concentración que lo que le llevó obtener la respuesta a su pregunta. Es por esta razón que las prácticas ágiles de desarrollo de software generalmente recomiendan que un equipo de trabajo esté en una sola sala donde todo el mundo tenga acceso directo a todos los demás miembros del equipo, inclusive para los clientes o sus representantes.
Las personas no son las únicas cosas que se mueven, la documentación lo hace también. Los requisitos pueden pasar de los analistas a los diseñadores, los documentos de diseño pueden moverse desde los diseñadores a los programadores, y luego el código se mueve de los codificadores al equipo de prueba, y así sucesivamente. Cada transferencia de un documento está llena de oportunidades para generar residuos. El mayor desperdicio del movimiento de documentación es que ésta no contiene toda la información que el receptor necesita conocer. Grandes cantidades de conocimiento tácito permanecen con el autor del documento y nunca se entregan al destinatario. En conclusión, el movimiento en general de un grupo a otro es una gran fuente de desperdicios en el desarrollo de software.

5.1.1.8. Defectos

La cantidad de residuos causados por un defecto es el producto entre impacto de éste y el tiempo que pasa desapercibido. Un defecto crítico detectado en tres minutos no es una gran fuente de desperdicios. Un defecto menor que no se descubre por semanas es una pérdida mucho más grande. La manera de reducir el impacto de los defectos es encontrarlos tan

pronto como se produzcan. Por lo tanto, la manera de reducir los residuos causados por defectos es realizar pruebas inmediatamente, integrar a menudo y lanzar a producción los productos tan pronto como sea posible.

5.1.2. MAPA DEL FLUJO DE VALOR

Trazar el flujo de valor es una buena manera de empezar a descubrir los residuos en un proceso de desarrollo de software. En una industria tras otra, el proceso de mapeo de una cadena de valor ha llevado invariablemente a la mejor comprensión de cómo los procesos internos de trabajo funcionan para satisfacer las necesidades del cliente.
La creación de un mapa de flujo de valor es un ejercicio que se puede realizar fácilmente al caminar alrededor de la organización. La idea es imaginar el recorrido de una solicitud de un cliente a través de cada paso del proceso. El objetivo es dibujar un gráfico de una solicitud promedio de un cliente, desde su llegada hasta su finalización. Trabajando con las personas involucradas en cada actividad, se podrá realizar un esquema de todos los pasos necesarios para cumplir con una solicitud, así como la cantidad promedio de tiempo que una solicitud consume en cada paso. Si una organización utiliza un proceso de desarrollo tradicional, entonces el mapa del flujo de valor será similar al de la Figura 5.1

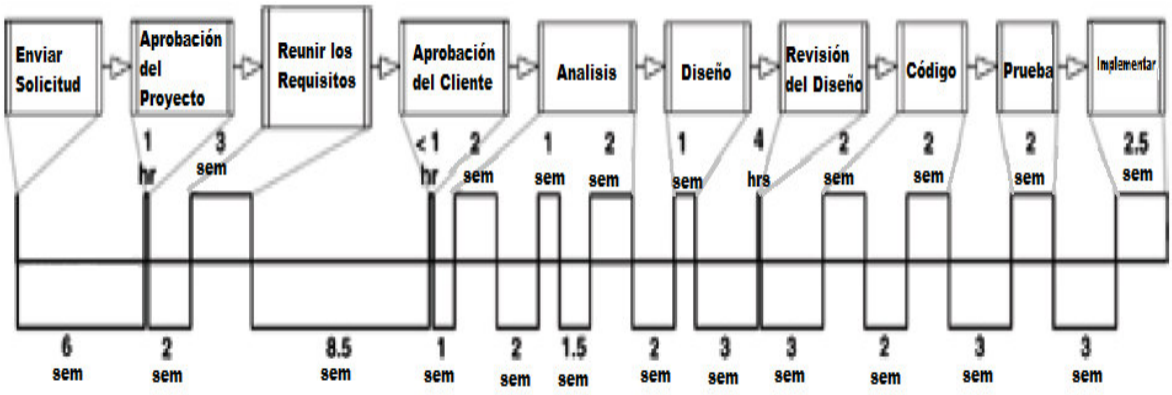


Figura 5.1 Mapa del flujo de valor tradicional

Este mapa muestra que un proyecto normal está listo para implementarse en un año, con alrededor de un tercio de tiempo dedicado a las actividades que aportan valor. El equipo de gestión revisa el proyecto cada doce semanas, por lo que se debe esperar un promedio de seis semanas antes de comenzar. Se debe competir por los recursos, lo que puede ser visto en los tiempos de espera para el análisis, diseño, codificación y pruebas. La aprobación por parte del cliente es muy lenta, tomando en promedio un par de meses. Esto probablemente se debe a que los clientes consideran la firma como un alto riesgo, ya que no conseguirán otra oportunidad de influir en lo que necesitan. La revisión del diseño toma tres semanas para programar y la codificación no comienza por otras tres semanas, ya que los desarrolladores están trabajando en otros proyectos. Los tiempos de prueba son cortos, indicando que algunos problemas se desarrollan a finales del proyecto. Sin embargo, se tarda casi seis semanas para implementar un sistema probado. Todo esto representa un tiempo largo.

5.1.2.1. Mapa de Flujo de Valor Ágil

Si suponemos que la organización representada en el mapa de la cadena de valor tradicional decide trasladarse a las prácticas ágiles. El mapa del flujo de valor en la Figura 5.1 puede generar el siguiente análisis:
El mapa de flujo de valor en cuestión indica que el proceso de aprobación se debe acortar, por lo que el equipo se compromete a reunirse semanalmente para tomar decisiones de aceptación o rechazo de nuevos requerimientos. El equipo decide que su máxima prioridad será la respuesta rápida a las solicitudes de los clientes, por lo que los miembros acuerdan aprobar sólo las solicitudes que se pueden manejar de inmediato y añadir personal o subcontratar para realizar las solicitudes adicionales. Se gestionará la disponibilidad del personal para que un equipo de diseño pueda ser asignado a los proyectos aprobados dentro de una semana. Todos los proyectos deben contar con el personal necesario en un plazo de tres semanas, con analistas y desarrolladores dedicados.
El mapa de flujo de valor inicial indica que la firma del cliente podría ser una fuente de irritación y retraso. Muestra que las revisiones del diseño deben moverse en línea con el desarrollo, ya que son una gran fuente de demoras. Por último indica que la planificación para la implementación debería ocurrir más temprano en el proceso. Desde que el equipo ha decidido trabajar sobre el desarrollo ágil, resolverá estos problemas al utilizar desarrollo incremental, reuniendo los requisitos según sea necesario, integrando las revisiones de diseño con la codificación y planificando de manera temprana para implementaciones regulares.
El mapa del flujo de valor ágil de este análisis sería similar al de la Figura 5.2. Este mapa muestra que con los cambios considerados, una petición de cliente típica debe moverse a través de la organización en unos tres meses, con la mayor parte de ese tiempo empleado realmente a agregar valor.

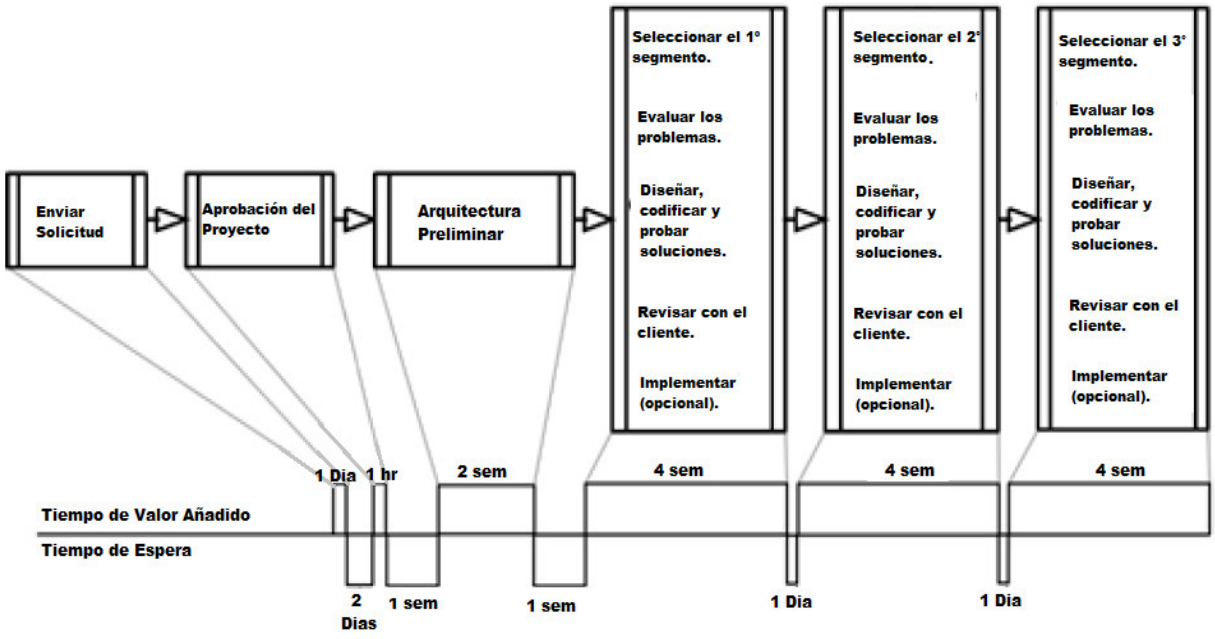


Figura 5.2 Mapa del flujo de valor ágil

Una vez que se tiene el esquema, se debe elegir las mejores oportunidades para aumentar el flujo y tiempo de valor agregado y luego de eso comenzar a trabajar. El siguiente paso es extender el mapa a los clientes, para intentar entender como éstos crean valor.
Muchas empresas han descubierto los beneficios del mapeo de la cadena de valor. Ayuda a las organizaciones a dar un paso hacia atrás y obtener una visión global de sus procesos. Es una herramienta para descubrir y eliminar actividades generadoras de residuos y agrupar las actividades que realmente crean valor en un flujo rápido que responde a la demanda del cliente. La razón por la que el mapeo de la cadena de valor es tan eficaz es porque centra la atención en los productos y su valor para los clientes, y no en las organizaciones, los bienes y las tecnologías.

5.2. SEGUNDO PRINCIPIO - AMPLIAR EL APRENDIZAJE

Pese a que los orígenes del pensamiento Lean yacen en la producción, el proceso de desarrollo es bastante diferente. Podría pensarse que el desarrollo es como crear una receta y la producción es como seguir la receta. Estas son diferentes actividades y deben llevarse a cabo con diferentes enfoques. El desarrollo es un proceso de aprendizaje que implica ensayo y error. En la fabricación, el objetivo es reproducir fielmente y repetidamente la "receta" creada en el proceso de desarrollo con un mínimo de variación.

5.2.1. CALIDAD EN EL DESARROLLO DE SOFTWARE

La calidad en el desarrollo de software da como resultado un sistema con integridad percibida e integridad conceptual. Integridad percibida significa que la totalidad del producto alcanza un equilibrio entre función, utilidad, confiabilidad y economía para satisfacer al cliente. Integridad conceptual significa que los conceptos centrales del sistema trabajan juntos de manera uniforme y coherente. Los clientes de un sistema de software percibirán integridad en un sistema si se soluciona su problema en un formato fácil de usar y de manera rentable. No importa si el problema no se comprende bien, si cambia con el tiempo, o es dependiente de factores externos; un sistema con integridad percibida es aquel que continuamente resuelve el problema de manera eficaz. Por lo tanto, la calidad en el diseño significa la realización de los fines o aptitud para el uso en lugar de la conformidad de los requisitos.

5.2.2. HACERLO BIEN LA PRIMERA VEZ

Con el fin de resolver problemas que no han sido resueltos antes, es necesario generar información. Para problemas complejos, el enfoque preferido para llegar a una solución es el método científico: observar, crear una hipótesis, diseñar un experimento para probar la hipótesis, ejecutar el experimento y ver si los resultados son consistentes con la hipótesis. Una de las características más interesantes del método científico es que si su hipótesis es siempre correcta, no se va a aprender mucho. La máxima cantidad de información se genera cuando la probabilidad de fallo es del 50%, no cuando las hipótesis son siempre correctas. Es necesario disponer de una tasa de fracaso razonable con el fin de generar una cantidad razonable de nueva información.
Hay dos escuelas de pensamiento en el desarrollo de software. Una es animar a los desarrolladores a asegurarse de que cada diseño y cada segmento de código sea perfectamente hecho la primera vez. La segunda escuela de pensamiento sostiene que es mejor tener pequeños y rápidos ciclos de prueba y corrección hasta asegurarse que el diseño y el código son perfectos. La primera escuela de pensamiento deja poco espacio para la generación de conocimiento a través de la experimentación, en su lugar propone que la generación de conocimiento debe ocurrir a través de la deliberación y revisión. El enfoque de "hacerlo bien la primera vez" puede trabajar efectivamente con problemas bien estructurados, pero el enfoque de "prueba y corrección" suele ser el mejor para problemas mal estructurados.
En una organización el objetivo debe ser equilibrar la experimentación con la deliberación y revisión. Para hacer esto debe considerarse como generar más conocimiento al menor costo

dependiendo de las circunstancias particulares. Por lo general, una combinación de experimentación, revisión por pares e iteración proporcionará los mejores resultados.

5.2.3. RETROALIMENTACION

Hay muchos acontecimientos imprevisibles en el desarrollo de software, por lo que es evidente la necesidad de contar con bucles de retroalimentación en el proceso.
En 1970, Winston Royce propuso un proceso de diseño de software secuencial que se parecía mucho a los procesos secuenciales de desarrollo de productos de la época. Abogó por la creación de documentación detallada en cada paso, pero también señaló que esperar hasta el final para poner a prueba el sistema no era práctico, ya que la información proporcionada por las pruebas era necesaria al principio del proceso de desarrollo. Por lo tanto, sugirió que era necesario en primera instancia un prototipo para obtener información de retroalimentación.[4] Ver Figura 6.1.

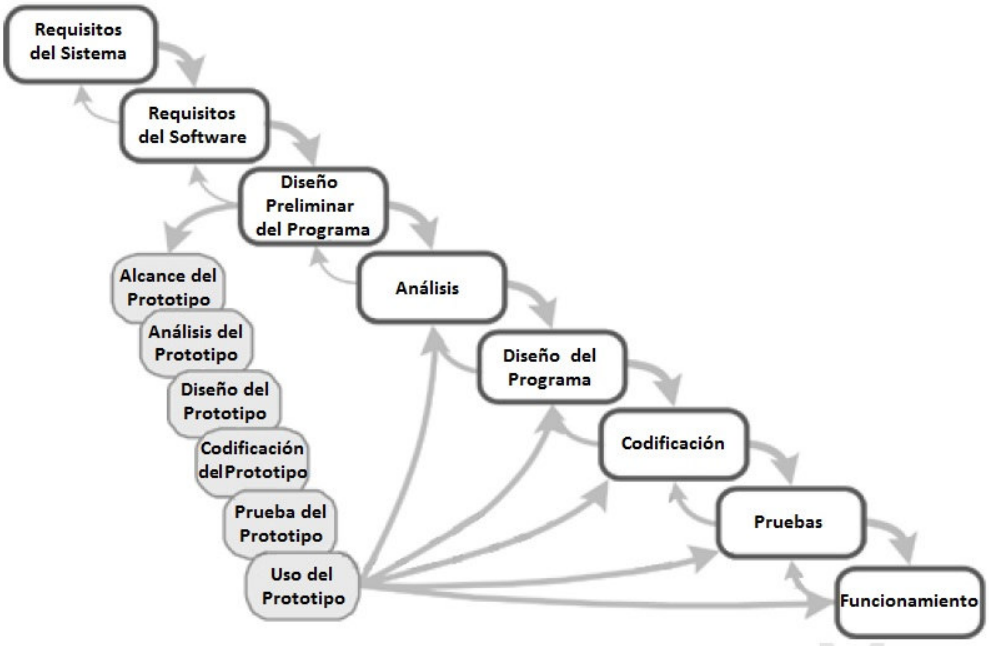


Figura 5.3 Propuesta Original de Royce de Modelo en “Cascada”

En la práctica, el modelo de desarrollo secuencial o “de cascada” no suele proveer mucha realimentación, sino que es generalmente considerado como un modelo de una sola pasada. Esto puede ser llamado un modelo determinista porque supone que los detalles de un proyecto son determinados al principio. Un modelo determinista se ve favorecido por las disciplinas de gestión de proyectos que tienen sus orígenes en la administración de contratos. El modelo de gestión de proyectos inspirado en contratos generalmente promueve un proceso de desarrollo secuencial con las especificaciones establecidas en el inicio del proyecto, certificadas por el cliente y con un proceso de autorización de cambios con tendencia a minimizarlos. De esta manera se genera la percepción de que estos procesos ofrecen mayor control y previsibilidad, a pesar de que los procesos de desarrollo secuencial con baja retroalimentación tienen un pésimo historial en ese sentido.[5]
El enfoque tradicional de gestión de proyectos a menudo considera a los bucles de realimentación como una amenaza, porque existe la preocupación de que el aprendizaje

involucrado en la retroalimentación podría modificar el plan predeterminado. En la visión convencional de gestión de proyectos, el alcance de la gestión, el costo y el cronograma son valuados de acuerdo al plan original. A veces esto se hace a costa de recibir y actuar en la retroalimentación que podría cambiar el plan; otras veces se hace a expensas de alcanzar el objetivo general de la empresa. Este modelo mental está tan arraigado en el pensamiento de gestión de proyectos que sus suposiciones subyacentes son raramente cuestionadas. Esto podría explicar porque es tan difícil abandonar el modelo de cascada de desarrollo de software.
Cuando una organización tiene desafíos de desarrollo de software, existe una tendencia a imponer un proceso más disciplinado en la organización. El concepto predominante de un proceso de desarrollo más disciplinado es aquel con el procesamiento secuencial más riguroso: Requisitos documentados de forma más completa, todos los acuerdos con el cliente son escritos, cambios controlados más cuidadosamente y cada requisito debe traducirse a código. Esto equivale a la imposición de controles deterministas adicionales en un entorno dinámico, alargando el ciclo de retroalimentación. Tal como predice la teoría de control, transforma una mala situación en una peor.
En la mayoría de los casos, el aumento de la retroalimentación, y no su disminución, es la manera más eficaz de hacer frente a proyectos de desarrollo y entornos problemáticos.

- En lugar de dejar que los defectos se acumulen, se debe ejecutar pruebas tan pronto como el código esté escrito.
- En lugar de añadir más documentación o planificación detallada, de recomienda tratar de comprobar las ideas mediante la escritura de código.
- En lugar de reunir más requisitos de los usuarios, procurar mostrarles una variedad de posibles pantallas de usuario y obtener su entrada.
- En lugar de estudiar más detenidamente que herramienta utilizar, se deben tomar las tres candidatas más prometedoras y probarlas.
- En lugar de tratar de encontrar la manera de convertir un sistema completo en una sola obra masiva, se pueden crear interfaces en red con el sistema anterior y probar la nueva idea.

Cuando las personas trabajan, deberían hacerlo para un cliente inmediato, es decir que, alguien en algún lugar debe estar dispuesto a hacer uso de los resultados de su trabajo. Los desarrolladores deben conocer a sus clientes inmediatos y tener maneras para que esos clientes proporcionen información periódica que pueda ser usada para retroalimentación. Cuando surge un problema, lo primero que se debe hacer es asegurarse que los bucles de realimentación estén en su lugar, es decir, asegurarse de que todo el mundo sepa quién es su cliente inmediato. El siguiente paso es aumentar la frecuencia de los ciclos de retroalimentación en las áreas problemáticas.

5.2.4. ITERACIONES

Si un fabricante quiere aplicar los principios de producción Lean, hay un punto de partida que siempre funciona, que es utilizar el flujo de inventario “Justo a Tiempo”. El simple hecho de trabajar para cumplir con los pedidos de los clientes, en lugar de trabajar para cumplir con un cronograma programado, impulsa una serie de mejoras adicionales. Una de las razones por las que el flujo “Justo a Tiempo” es tan eficaz es que se requiere mejorar la

comunicación de trabajador a trabajador y abordar problemas de calidad tan pronto como aparezcan.
En el desarrollo concurrente de productos existe un punto de partida universal equivalente: dirigir el esfuerzo a través de prototipos para alcanzar hitos cercanos entre sí. Un prototipo sincroniza esfuerzos hacia un objetivo bien definido a corto plazo sin necesidad de programación detallada. Estos hitos hacen posible el desarrollo concurrente de productos, ya que proporcionan un punto focal alrededor del cual puede y debe producirse la comunicación interfuncional. Los prototipos también proporcionan retroalimentación temprana ante problemas de diseño y preferencias de los clientes.
Para las metodologías ágiles de desarrollo de software existe también un punto de partida universal: las iteraciones. Una iteración es un incremento de software útil que está diseñado, programado, probado, integrado y entregado en un corto periodo de tiempo fijado. Es muy similar a un prototipo en el desarrollo de productos, excepto que una iteración produce una porción funcional del producto final. Este software se puede mejorar en futuras versiones, pero es código funcional, probado e integrado desde el principio. Las iteraciones proporcionan un aumento sustancial en la retroalimentación sobre el desarrollo secuencial de software, lo que proporciona comunicación mucho más amplia entre los clientes/usuarios y desarrolladores. Los problemas de diseño están expuestos temprano, por lo tanto cuando ocurran cambios, la tolerancia a estos cambios ya se encuentra integrada en el sistema.

5.2.4.1. PLANIFICACION DE ITERACIONES

Dentro de cada iteración, la idea es implementar un conjunto coherente de características. Una característica es algo que proporciona valor significativo para el cliente, pero que es lo suficientemente pequeño como para que el equipo pueda estimar confiablemente el esfuerzo necesario para entregarlo. Si una característica no puede ser realizada en una sola iteración, debe ser dividida en varias más pequeñas. Las características provienen de los clientes o sus representantes en forma de casos de uso, historias o elementos pendientes.[6]
Al comienzo de cada iteración, una sesión de planificación tiene lugar, en la cual el equipo de desarrollo estima el nivel de dificultad de las características consideradas, y los clientes o sus representantes deciden cuales son las más importantes de acuerdo a su costo estimado. Las características de mayor prioridad deberían desarrollarse primero con el fin de entregar los elementos que proporciones el mayor valor de negocio en primer lugar.
Una iteración debe tener un margen de tiempo fijado. Algunas personas sugieren mantener todas las iteraciones de la misma longitud para establecer un ritmo. Otros proponen variar la longitud de las iteraciones en base a las circunstancias locales.
El margen de tiempo de las iteraciones debería ser suficiente para soportar un ciclo significativo de diseño, construcción y prueba, y lo suficientemente corto como para proporcionar retroalimentación frecuente de los clientes para los cuales el sistema está en marcha. Algunas personas sienten que un mes de margen de tiempo es ideal. Otros sugieren un par de meses de plazo. Algunas empresas utilizan de 6 a 10 semanas, pero en ese tiempo se acoplan compilaciones diarias y pruebas semanales extensas.
El equipo de desarrollo debe ser libre de aceptar sólo la cantidad de trabajo para una iteración que los miembros del equipo crean posible completar dentro del margen de tiempo establecido. Los clientes probablemente desearán implementar iteraciones con muchas características, pero es importante resistir la tentación de ser complaciente a costa de crear expectativas poco razonables. Si las iteraciones son cortas y la entrega es fiable, los clientes

deben estar dispuestos a esperar para la próxima iteración. Si un equipo de desarrollo se compromete en demasía, lo que a menudo ocurre con los equipos sin experiencia, lo mejor es entregar algunas de las funciones a tiempo en lugar de entregar todas tarde.

5.2.4.2. COMPROMISO DEL EQUIPO

Un equipo de proyecto puede evaluar una lista de características y, con un poco de investigación, llegar a una buena idea de lo que puede hacer en un par de semanas o un mes. Sin embargo, no se puede esperar que un equipo establezca y cumpla con los objetivos dentro del marco de tiempo establecido sin apoyo organizativo.[7]

- El equipo debe ser pequeño y estar conformado por personal con la experiencia necesaria. Algunos miembros del equipo deben tener experiencia en el dominio y otros en cada tecnología crítica.
- El equipo debe tener suficiente información sobre las características solicitadas para ser capaz de decidir lo que es factible lograr en el margen de tiempo establecido.
- El equipo debe estar seguro de poder obtener los recursos que sean necesarios.
- Los miembros del equipo deben tener la libertad, el apoyo y la habilidad para encontrar la manera de cumplir sus compromisos.
- El equipo debe tener o crear el entorno básico para la buena programación: proceso de generación automática, prueba automatizada, estándares de codificación, herramientas de control de versiones, etc.

Una buena planificación de iteraciones ofrece a los clientes una manera de pedir las características que son importantes para ellos y crea un ambiente motivador para el equipo de desarrollo. Lo mejor de estos beneficios es que se alimentan en base a éxitos. Al ver los clientes las características que consideran de más alta prioridad implementadas en código, comienzan a creer que el sistema será real y comienzan a imaginar qué se puede hacer para mejorarlas. Los clientes se sienten más cómodos al saber que las características previstas para futuras iteraciones se entregarán en realidad. Al mismo tiempo, los desarrolladores obtienen un sentido de logro, y ya que los clientes comienzan a apreciar su trabajo, son incluso más motivados para satisfacerlos.

5.2.4.3. CONVERGENCIA

Las iteraciones suenan como una buena idea, sin embargo, hay una renuencia considerable a utilizarlas. La razón detrás de esto a menudo puede atribuirse al temor de que el esfuerzo puesto en el desarrollo de software no sea convergente. Existe la preocupación de que el proyecto continuará indefinidamente si no se tiene un punto de parada predefinido.[8] Esta es una preocupación válida, ya que es difícil estar seguro que cualquier sistema con ciclos de retroalimentación podrá converger en una solución.
Una situación de negocios fluida podría enviar señales impredecibles y constantemente cambiantes al proceso de desarrollo de software. Esto significaría que la retroalimentación cambia tan rápido que el sistema no dispone del tiempo para completar una respuesta antes de que se determine ir en la dirección opuesta.
Un proceso iterativo de desarrollo de software limita las peticiones de cambios de características al inicio de cada bucle. Durante la iteración, el equipo se concentra en la

entrega de las características que se acordaron al principio de la misma. Si las iteraciones son cortas (entre dos y cuatro semanas), el bucle de retroalimentación es todavía bastante corto.
Retrasar la respuesta a la retroalimentación debe ser manejado con cuidado, ya que largos tiempos de demora tienden a causar oscilaciones del sistema. La convergencia requiere pequeños ajustes frecuentes. Por ejemplo, un control de velocidad crucero ajusta el acelerador ligeramente solo cuando el automóvil cae por debajo de la velocidad deseada. Del mismo modo, si el software es entregado en incrementos pequeños y frecuentes, el cliente puede ver cómo va creciendo el valor y hacer ajustes sobre una base regular. La entrega de incrementos grandes con poca frecuencia probablemente producirá oscilaciones.

5.2.4.4. ALCANCE NEGOCIABLE

Una buena estrategia para lograr la convergencia es trabajar primero en los principales ítems que son prioridad, dejando los elementos de baja prioridad en la lista de tareas pendientes. Al ofrecer las características prioritarias en primer lugar, lo más probable es que se entregue la mayor parte del valor mucho antes que se complete la lista de deseos del cliente. Sin embargo, si se está trabajando bajo la expectativa de que el desarrollo no está completo hasta lograr un alcance fijo y detallado, entonces el sistema puede de hecho, no converger. Por lo tanto, es mejor evitar esta expectativa, ya sea indicando en primera instancia que el alcance es negociable, o mediante la definición del alcance en un nivel alto, por lo que en detalle es negociable. Con alcance negociable, el desarrollo iterativo generalmente converge.
Un estudio del StandishGroup determinó que el 45 por ciento de las características de un sistema típico no se utilizan nunca y que el 19 por ciento se utilizan raramente.[9] Puesto que los clientes a menudo no saben exactamente lo que quieren al principio del proyecto, tienden a pedir todo lo que ellos consideran que podría ser necesario, especialmente si piensan que tendrán una sola oportunidad de hacerlo. Esta es una de las mejores formas conocidas para aumentar el alcance de un proyecto mucho más allá de lo necesario para cumplir con la misión general del proyecto.
Si se deja que los clientes pidan sólo sus necesidades de mayor prioridad, se las debe entregar rápidamente para continuar luego con la siguiente prioridad más alta. De esta manera se crea una tendencia a tener listas cortas de lo que es realmente importante. Por otra parte, se puede responder a las circunstancias cambiantes. Por lo tanto, suele ser buena idea trabajar en base a una lista de prioridades ordenadas.
Este enfoque de gestión de proyectos puede parecer que da lugar a resultados impredecibles, pero en realidad es todo lo contrario. Una vez establecido un historial de entrega de software funcional, es fácil proyectar la cantidad de trabajo que se llevará a cabo en cada iteración a medida que el proyecto avanza. Mediante el seguimiento de la velocidad del equipo, se puede predecir a partir de trabajos anteriores la cantidad de trabajo que probablemente se llevará a cabo en el futuro. Las mediciones de velocidad son herramientas mucho más precisas que los controles basados en el alcance, ya que miden la cantidad de tiempo que en realidad tomó entregar el código completo y probado al final de cada iteración.

5.2.5. SINCRONIZACION

Las iteraciones se planifican seleccionando las características o funciones que son importantes para los clientes, y si hay varios equipos involucrados, por lo general dividen el trabajo de acuerdo a las características a desarrollar. Uno de los problemas del enfoque basado en características para el desarrollo de software es que éstas probablemente involucran diferentes áreas del código. Tradicionalmente, la integridad de un módulo era asegurada teniendo un solo desarrollador asignado al trabajo, que entendía claramente las características del módulo. Los métodos más ágiles recomiendan la propiedad común del código, aunque el Desarrollo Basado en Características (DBC) mantiene la propiedad individual de los módulos o clases.[10] Dado que las características individuales requieren varias clases diferentes a ser modificadas, el Desarrollo Basado en Características forma equipos integrados por los propietarios de las clases más relevantes.
Cuando varias personas están trabajando en lo mismo, se produce una necesidad de sincronización. En el DBC es necesaria la sincronización de varias personas que trabajan en una característica, mientras que la propiedad del código común requiere que varias personas que trabajan en el mismo segmento de código estén sincronizadas. La necesidad de sincronización es fundamental para cualquier tipo de desarrollo complejo.
El mismo problema se produce en el diseño de automóviles. Un ligero cambio en una pieza para mejorar la aerodinámica podría tener impacto en la forma o disposición de otros componentes. Cuando las cosas se complican en el diseño de automóviles, no se puede construir una nueva maqueta para ver como encajan realmente las cosas. Toyota construye muchos más prototipos que la mayoría de los fabricantes de automóviles, ya que son una manera efectiva para sincronizar rápidamente los esfuerzos de muchas personas.

5.2.5.1. SINCRONIZAR Y ESTABILIZAR

En un entorno de desarrollo de software con propiedad colectiva del código, la idea es construir el sistema cada día, después de que una pequeña porción de trabajo ha sido realizada por cada uno de los integrantes del equipo de desarrollo. Diariamente, los desarrolladores revisan el código fuente de un sistema de gestión de configuración, hacen cambios, los prueban para verificar si alguien ha realizado cambios al mismo código y, de ser así, se comprueba si hay conflictos; luego revisan el nuevo código. Al final del día, una construcción tiene lugar, seguido por un conjunto de pruebas automatizadas. Si lo construido funciona y se pasan las pruebas, los desarrolladores se han sincronizado.
Hay muchas variaciones sobre este tema. Una compilación puede ocurrir cada pocos días, o puede ejecutarse cada vez que se ingresa nuevo código. Construcciones frecuentes son mejores, ya que proporcionan retroalimentación mucho más rápida. El proceso de construcción y prueba debe ser automatizado. Si no es así, el proceso en sí introducirá errores y la cantidad de trabajo manual impedirá construcciones suficientemente frecuentes.
A veces la construcción abarca todo el sistema, otras veces solo se construyen subconjuntos del mismo, porque el sistema completo es demasiado grande. También en ocasiones, todo un conjunto de pruebas se ejecuta, y en otras, sobre todo cuando las pruebas son manuales, solo se ejecutan algunas. El principio general es que si la construcción y prueba demora demasiado, el conjunto no va a ser utilizado, por lo tanto se debe invertir en hacerlo más rápido. Esto proporciona un sesgo hacia construcciones más

frecuentes con pruebas menos exhaustivas, pero conservando la importancia de ejecutar todas las pruebas diariamente o semanalmente.

5.2.5.2. MATRIZ

Un enfoque más tradicional de sincronizar varios equipos es esbozar una arquitectura global y luego dividir en subsistemas o componentes separados con su propio equipo de desarrollo. Este enfoque es particularmente apropiado cuando los diferentes equipos no se encuentran en el mismo lugar, ya que les permite abordar su trabajo con un mínimo de comunicación con otros equipos. El problema, por supuesto, se produce en las interfaces. Cuando los componentes de los distintos equipos tienen que trabajar juntos, suele ser necesaria una poderosa comunicación para resolver los problemas de diseño que puedan surgir. Por otra parte, si los equipos ya han desarrollado sus subsistemas, es probable que no estén dispuestos a modificar el trabajo ya hecho.
Por lo tanto, el enfoque matricial se inicia mediante el desarrollo de las interfaces y luego los subsistemas. Todos los puntos de interacción entre equipos deben establecerse al principio y los equipos deben ser asignados a cada uno de estos puntos de interacción. Las interfaces deben ser desarrolladas primero, y luego de que estén funcionando, los equipos pueden trabajar de manera razonablemente independiente en el desarrollo de sus subsistemas, pero deben integrar regularmente su código dentro del sistema completo para asegurarse de que la interfaz continúa trabajando.
Este enfoque fue utilizado por Motorola para diseñar un nuevo sistema de comunicación.[11] Equipos de todo el mundo estaban involucrados y cada uno se encargó de desarrollar el software en una única pieza de hardware. Antes de que los equipos iniciaran su diseño del subsistema, se reunieron en un solo lugar para estudiar la estructura general y definir las interacciones entre los dispositivos. Cada enlace entre dispositivos, llamado estrato, fue identificado, y un equipo formado por una persona de cada equipo de subsistema fue asignado a cada estrato. Esto es ilustrado en la Figura 6.2, que muestra los estratos entre los dispositivos A, B, C, D y E.

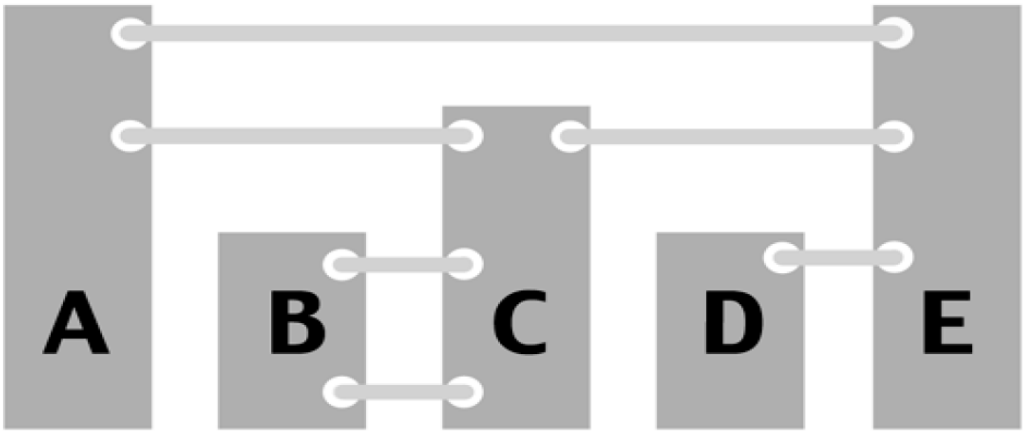


Figura 5.4 Implementar Interfaces Primero

La belleza de este enfoque es que las zonas de más alto riesgo que pueden causar los mayores retrasos y crear los mayores problemas de comunicación fueron las interacciones entre equipos, las cuales fueron resultas en el inicio del proyecto. La parte más fácil, la integración de los equipos, se guardó para más tarde en el proyecto. Esta técnica

proporciona la mejor sincronización a través del proyecto, debido a que un equipo podría integrar código dentro de la estructura general regularmente, asegurándose de que todo lo que se hizo desde adentro no compromete el sistema global.

5.2.6. DESARROLLO BASADO EN CONJUNTOS

En el desarrollo basado en conjuntos, la comunicación está basada en las restricciones y no en las opciones. Esta resulta ser una forma muy potente de comunicación que requiere significativamente menos datos para transmitir mucha más información. Además, al hablar de las restricciones en lugar de las opciones, se aplaza la toma de decisiones hasta el momento en que realmente son necesarias, es decir, hasta el último momento responsable.
Consideremos cómo la comunicación basada en restricciones puede acelerar el desarrollo de productos a gran escala. DurwardSobek estudió los enfoques de desarrollo de productos de Toyota y Chrysler y descubrió que una disciplina de ingeniería primordial de Toyota es mantener y hacer referencia a listas de verificación (*checklists*), dónde se registra las compensaciones y limitaciones conocidas.[12]
Por ejemplo, un ingeniero encargado de definir el estilo puede desear un guardabarros trasero con un nuevo aspecto dramático. Sin embargo, el ingeniero de fabricación podría sospechar que el nuevo diseño será difícil de fabricar. En lugar de expresar una duda vaga, el ingeniero de fabricación podría enviar al encargado de diseño una lista de verificación mostrando el tiempo que se necesita para realizar los paneles de la carrocería con ciertas características y detallando los límites de las mismas. El checklist no es necesariamente una lista; a menudo es un gráfico de las condiciones límite, similar a la *Figura 6.3.* El ingeniero encargado del estilo podría examinar el checklist junto con muchos otros similares y llegar a dos o tres diseños que tienen todas las limitaciones en consideración.

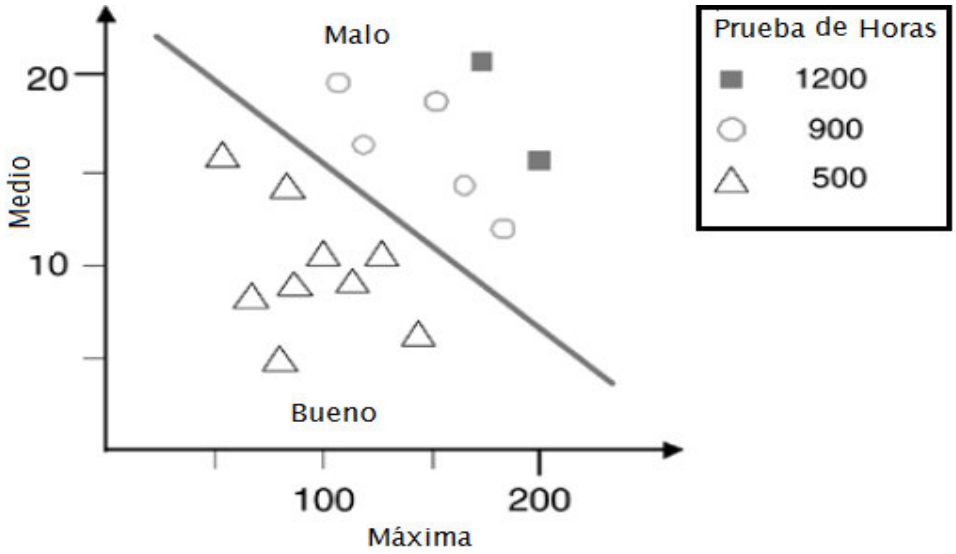


Figura 5.5 Checklist: relación de la deformación de la sección transversal del guardabarros trasero.

Toyota explora un gran número de conceptos al comienzo de un programa de fabricación de un vehículo, gastando significativamente más recursos que otros fabricantes de automóviles. Se mantiene una gran cantidad de opciones a lo largo del proceso de desarrollo y se produce un extraordinario número de prototipos de subsistemas y modelos de los vehículos.

Las dimensiones finales se fijan mucho más adelante en el proceso de desarrollo de lo que es común entre otros fabricantes de automóviles y las especificaciones finales son liberadas a los proveedores lo más tarde posible en el proceso. La calidad, popularidad y rentabilidad de los automóviles que produce indican que el proceso de desarrollo de Toyota es altamente eficaz.[13]

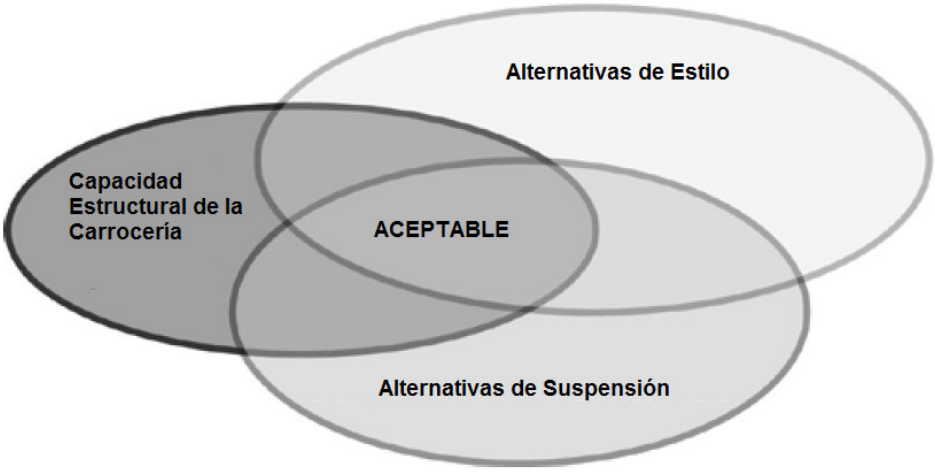


Figura 5.6 Desarrollo Basado en Conjuntos

Para aplicar el desarrollo basado en conjuntos al software se deben:

- Desarrollar múltiples opciones
- Comunicar las restricciones
- Dejar emerger las soluciones

5.2.6.1. Desarrollar Múltiples Opciones

Cuando se presenta un problema difícil, se debe intentar desarrollar un conjunto de alternativas de solución al problema, ver que tan bien funcionan realmente y luego fusionar las mejores características de las soluciones o elegir una de las alternativas. Podría parecer un desperdicio desarrollar múltiples soluciones para un mismo problema, pero el desarrollo basado en conjuntos puede llevar a mejores soluciones más rápidamente, como en el siguiente ejemplo:

Desarrollo de un Sitio Web Basado en Conjuntos
Cuando no se está de acuerdo sobre la forma de estructurar el sitio Web, se crean dos o tres versiones con diferentes rutas (paths) y diseños de páginas (layouts). A continuación se hacen pruebas de usabilidad con varios usuarios finales. Se observará que nunca hay un diseño que se destaca por encima de los demás. En su lugar, encontramos que algunas de las características de cada diseño son buenas y otras son más bien pobres. Se reúne las mejores características de todas las opciones y se vuelve a probar. Invariablemente se obtiene una mejor facilidad de uso con ésta combinación y por lo tanto se podría considerar que es la mejor manera de diseñar un sitio web.

El desarrollo basado en conjuntos no reemplaza al desarrollo iterativo, sino que añade una nueva dimensión. Durante las primeras iteraciones, se desarrollan múltiples opciones para las características claves y en iteraciones posteriores se fusionan o se comprimen a una sola elección.

5.2.6.2. Comunicar Restricciones

El desarrollo basado en conjuntos significa que se comunican limitaciones, no soluciones. En la superficie esto parece ser lo contrario al uso de un enfoque iterativo. Se supone que para producir trabajo se debe desplegar código en cada iteración, por lo tanto una iteración puede parecer una solución basada en un punto, que es lo contrario al desarrollo basado en conjuntos.
Pensar en una iteración como una solución basada en un punto es una mala interpretación del desarrollo iterativo. En una iteración se implementa solo la cantidad mínima de funcionalidad necesaria para demostrar los conceptos centrales de esa iteración. Por ejemplo, no se comienza con un diseño de una base de datos completa en la primera iteración, en lugar de esto se utilizan capas simples para lidiar con los subconjuntos de características. El diseño evolucionará, y en ese sentido, la primera iteración es un prototipo de una pieza del diseño general.
Una iteración se debe considerar como una demostración de una solución posible y no como la única solución. Las primeras iteraciones deben dejar un amplio margen para la aplicación del resto del sistema de muchas maneras posibles. A medida que las iteraciones progresan y más opciones son presentadas, el espacio de diseño debería reducirse gradualmente.

5.2.6.3. Dejar Emerger las Soluciones

Comunicar restricciones es de gran utilidad al abordar un problema particularmente difícil, debido a que ayuda a asegurar que la solución es elaborada por todos los interesados. A medida que el grupo se enfrenta con el problema, se debe resistir la tentación de saltar a una solución. Se debe mantener visibles las restricciones del problema de modo que el equipo pueda descubrir la intersección del espacio de diseño que funcione para todos los interesados.

5.3. TERCER PRINCIPIO - DECIDIR LO MÁS TARDE POSIBLE

5.3.1. DESARROLLO SIMULTÁNEO

Cuando una chapa es moldeada para formar parte de una carrocería de automóvil, una enorme prensa mecánica presiona el metal dándole forma. Esta máquina tiene una matriz metálica que hace contacto con la lámina de chapa y presiona dándole forma de guardabarros, puerta u otro panel de carrocería de un automóvil. El diseño y corte de las matrices en la forma adecuada representa la mitad de la inversión de capital para el programa de desarrollo de un nuevo automóvil y forma parte del camino crítico. Si se comete un error en la producción de las matrices, la totalidad del programa de desarrollo sufre un gran retraso. Si hay una cosa que los fabricantes quieren hacer bien, es el diseño de las matrices y cortes.

El problema es que mientras el desarrollo avanza, los ingenieros siguen haciendo cambios al automóvil y estos cambios normalmente afectan el diseño de las matrices. Independientemente de que los ingenieros intenten congelar el diseño, no están en condiciones de hacerlo. En Detroit, en la década de 1980, el costo de los cambios en el diseño era del 30 al 50 por ciento del costo total del molde, mientras que en Japón fue del 10 al 20 por ciento. Estas cifras parecen indicar que las empresas japonesas tienen que haber sido mucho mejores para prevenir los cambios después de que las especificaciones de una matriz fueron lanzadas a la producción y luego a su comercialización.

La estrategia de EE.UU. para la fabricación de una matriz era esperar hasta que se congelen las especificaciones y luego enviar el diseño final de la herramienta a los fabricantes de moldes, lo que desencadenaba el proceso del pedido del bloque de acero y cortarlo. Cualquier cambio representaba un difícil proceso de aprobación de las modificaciones. Tomaba alrededor de dos años a partir del pedido del acero hasta que la matriz fuera usada en la producción. En Japón, sin embargo, se realizaba el pedido a los fabricantes de moldes y estos buscaban los bloques e iniciaban el proceso de corte al mismo instante en el que el proceso de producción de un automóvil se iniciaba. Esto se conoce como **desarrollo simultaneo**.

Los ingenieros japoneses saben bien lo que involucra realizar una matriz para los paneles de las puertas y están en constante comunicación con el cuerpo de ingenieros.[14] Ellos prevén la solución final y están especializados en técnicas para hacer cambios menores cuando el desarrollo ha avanzado, dejando más material donde los cambios son más probables. La mayoría de los ingenieros son capaces de adaptar la ingeniería de diseño a medida que evoluciona. En el caso de un posible error, una nueva matriz se puede confeccionar mucho más rápido ya que todo el proceso es más sencillo.

Las automotrices japonesas no congelan puntos de diseño hasta el final del proceso de desarrollo, permitiendo que la mayoría de los cambios ocurran mientras la posibilidad de cambios siga abierta. En comparación con las prácticas de congelamiento de diseño inicial de los americanos en la década de 1980, los fabricantes japoneses gastaron el tercio de la cantidad de dinero en cambios y tienen los mejores diseños. Las matrices japonesas demostraban una determinada tendencia de requerir un menor número de ciclos de estampados por parte, creando un significativo ahorro de producción.[15]

La sorprendente diferencia en el tiempo de puesta en el mercado y el gradual éxito comercial de los automóviles japoneses llevaron a las empresas automotrices americanas a

adoptar las prácticas de desarrollo simultáneo en la década de 1990, y hoy la brecha en el rendimiento de desarrollo de productos se ha reducido notablemente.

5.3.2. DESARROLLO SIMULTÁNEO DE SOFTWARE

La programación se parece mucho al proceso de moldeado. Los riesgos son altos y los errores pueden ser costosos, y de igual manera que en el desarrollo secuencial, establecer los requisitos antes de iniciar el desarrollo es comúnmente considerado como una forma de protegerse contra los errores graves. El problema con el desarrollo secuencial es que obliga a los diseñadores a tener profundidad inicial en lugar de amplitud preliminar sobre el enfoque del diseño. Esta profundidad inicial fuerza a tomar decisiones de bajo nivel antes de experimentar las consecuencias de las decisiones de alto nivel. Los errores más costosos se cometen al no considerar algo importante al principio. La forma más fácil de cometer un error es profundizar en los detalles demasiado rápido. Una vez establecido el camino detallado, no se puede volver atrás y es difícil percatarse de que se debe hacerlo. Cuando se cometen grandes errores lo mejor es estudiar el panorama y retrasar las decisiones detalladas.

El desarrollo de software simultáneo por lo general toma una forma de desarrollo iterativo. Este es el enfoque preferido cuando las apuestas son altas y la compresión del problema está evolucionando.

El desarrollo simultáneo permite tener un primer acercamiento amplio y descubrir los grandes y costosos problemas antes que sea tarde. El paso del desarrollo secuencial al desarrollo simultáneo significa empezar a programar las características de mayor valor tan pronto como se determine un diseño conceptual de alto nivel, incluso mientras se están investigando los requisitos detallados. Esto puede sonar contradictorio, pero debe ser considerado como un enfoque exploratorio, que permite aprender mediante la prueba de una variedad de opciones antes de tomar un rumbo que limite las características menos importantes.

Además de proporcionar un seguro contra errores costosos, el desarrollo simultáneo es la mejor manera de hacer frente a las necesidades cambiantes, porque además de ser postergadas las grandes decisiones mientras se tienen en cuenta otras opciones, las pequeñas decisiones se posponen también. Cuando el cambio es inevitable, el desarrollo simultáneo reduce el tiempo de entrega y el costo total, mientras se mejora el rendimiento del producto final.

Pese a estas grandes ventajas, se debe tener en cuenta al iniciar la programación, que sin experiencia y colaboración asociada es poco probable llegar a mejores resultados. Hay algunas habilidades fundamentales que se deben dominar para trabajar en desarrollo simultáneo.

En el desarrollo secuencial, los ingenieros fabricantes de matrices de automóviles americanos trabajaban lejos de los ingenieros automotrices. Del mismo modo, los programadores del modelo secuencial tienen poco contacto con las necesidades de los clientes y usuarios, así también como con los analistas que recogen los requisitos. Para aplicar el desarrollo simultáneo en la industria automotriz se tenían que realizar dos cambios: los ingenieros que confeccionan las matrices necesitaban tener la experiencia para anticiparse a las necesidades del nuevo diseño y saber dónde se tenía que realizar el corte del acero, y a su vez ellos también tenían que colaborar estrechamente con el cuerpo de ingenieros.

Del mismo modo, el desarrollo de software en simultáneo requiere desarrolladores con suficiente experiencia en el dominio para anticiparse y saber cómo manejar los probables nuevos diseños. Además deben tener una estrecha colaboración con los clientes y analistas que se dedican a diseñar los sistemas para resolver los problemas emergentes.

5.3.3. COSTOS EN ASCENSO

Un software es diferente a la mayoría de los productos, ya que se espera que sea actualizado con regularidad. En promedio, más de la mitad del trabajo de desarrollo de un sistema de software se realiza después que se vende o luego de ser puesto en proceso de producción.[16] Además de los cambios internos, los sistemas de software están sujetos a un entorno cambiante, como ser un nuevo sistema operativo, un cambio en las bases de datos, un cambio en el cliente usado por la Interfaz Gráfica de Usuario (GUI), una nueva aplicación que utiliza la misma base de datos, y así sucesivamente. Se estima que la mayoría de los sistemas de software cambien regularmente durante su vida útil, y de hecho, una vez que se detienen las actualizaciones es el momento en que los sistemas están llegando al final de su vida útil. Esto plantea una nueva categoría de residuos: los originados por el software que es difícil de modificar.

En 1987 Barry Boehm escribió “El costo de buscar y encontrar un problema en un software después de ser entregado es 100 veces mayor que encontrar y corregir el problema en las fases iniciales de diseño”.[17] Esta observación se convirtió en el motivo de realizar un profundo análisis y diseño de requisitos iniciales, a pesar de que Boehm apoyó el desarrollo incremental sobre “el desarrollo de productos en un solo paso”[18]. En 2001, Boehm señaló que el factor de aumento de costos para pequeños sistemas puede ser de 5:1 a 100:1, e incluso en sistemas grandes, con buenas prácticas arquitectónicas se puede reducir significativamente el costo del cambio mediante el aislamiento de características que son propensas a cambiar en áreas pequeñas y bien encapsuladas.[19]

Solía haber un factor de aumento de costos similar, pero más dramático, para el desarrollo de productos. Se estima que un cambio que se realiza después iniciada la producción va a costar 1000 veces más que si el cambio se hubiese hecho en el diseño original. La idea de que el costo de los cambios se intensificaba a medida que avanza el desarrollo contribuyó en gran medida a la estandarización del proceso de desarrollo incremental. Nadie parecía reconocer que el proceso secuencial podría ser en realidad la causa de la elevada relación de escalada. Sin embargo, a medida que el desarrollo en simultáneo sustituyó al desarrollo secuencial en Estados Unidos en la década de 1990, la discusión sobre la escalada de costos se alteró definitivamente. Ya no se discutió sobre el costo de un cambio una vez avanzado el desarrollo. En vez de esto, el debate se centró en cómo reducir la necesidad de cambio a través de la ingeniería concurrente o en simultáneo.

No todo cambio es igual. Hay algunas pocas decisiones sobre las arquitecturas básicas que se necesitan saber para el inicio del desarrollo, y éstas permiten solucionar las limitaciones que pueda tener un sistema durante su vida útil. Ejemplos de éstos pueden ser la elección del lenguaje de programación, decisiones en las capas arquitectónicas, o la opción de interactuar con una base de datos existente utilizando también otras aplicaciones. Este tipo de decisiones puede tener relación con el aumento de los costos debido a que son de importancia, al punto que se debe centrar la atención en reducir al mínimo el número de restricciones de alto riesgo, teniendo una primera visión sobre estas decisiones.

La mayor parte del cambio de un sistema no tiene que tener un factor de escalada de alto costo; es el enfoque secuencial el que hace que el costo de la mayoría de los cambios aumente exponencialmente a medida que se mueven a través del proceso. El desarrollo secuencial hace hincapié en conseguir que todas las decisiones sean tomadas lo más pronto posible, por lo que el costo de todos los cambios puede ser muy alto. El Diseño en Simultáneo pospone las decisiones la mayor cantidad de tiempo posible. Esto tiene cuatro efectos:

Reduce el número de limitaciones de gran relevancia.
Da un primer acercamiento amplio a las decisiones de alto riesgo, por lo que es más probable que se realicen correctamente.
Aplazar la mayoría de las decisiones, para reducir las necesidades de cambio.
Disminuir de manera radical el factor que produce el aumento de los costos para la mayoría de los cambios.

Es engañoso tener un solo factor de escalada de costos o una sola curva.[20] En lugar de mostrar un gráfico que modela una única directriz para todos los cambios, sería más apropiado mostrar un gráfico que presente al menos dos curvas de aumento de costos, como se muestra en la Figura 7.1. El objetivo del desarrollo ágil es mover tantos cambios como sea posible de la curva superior a la curva inferior.

Figura 5.7 - Dos curvas de aumento de costos

El desarrollo de software *Lean* tiene la particularidad de retrasar, congelando todas las decisiones de diseño todo el tiempo que sea posible, ya que es más fácil cambiar una decisión que aún no se ha concretado. El Desarrollo de Software Lean hace hincapié en un desarrollo sólido, tolerante a cambios y que admita la inevitabilidad del cambio, estructurando el sistema de modo que se pueda adaptar fácilmente a los probables tipos de variaciones.
La razón principal del cambio de un software a lo largo de su ciclo de vida es que el proceso empresarial en que es utilizado evoluciona con el tiempo. Algunos ámbitos evolucionan más

rápidamente que otros y algunos pueden ser esencialmente más estables. No es posible desarrollar de manera flexible para adaptar cambios arbitrarios a bajo costo. La idea es fomentar la tolerancia al cambio en el sistema a lo largo de las dimensiones del dominio que probablemente cambien. Observar donde se producen cambios durante el desarrollo iterativo proporciona un buen indicio de donde es probable que se necesite flexibilidad en el futuro. Si ciertos tipos de cambios son frecuentes durante el desarrollo, se puede suponer que continúen ocurriendo cuando el producto esté finalizado. El secreto es saber lo suficiente sobre el ámbito para mantener la flexibilidad, pero evitar hacer las cosas más complejas de lo que deben ser.
Si un sistema es desarrollado permitiendo que el diseño surja través de las iteraciones, el diseño será sólido, adaptándose más fácilmente a los tipos de cambio que se producen durante el desarrollo. Más importante aún, la capacidad de adaptación se desarrollará en el sistema, de modo que a medida que se producen cambios durante su entrega, estos pueden ser incorporados más fácilmente. Por otro lado, si el sistema se construye centrándose en conseguir todo desde el principio con el fin de reducir los costos de los cambios posteriores, su diseño probablemente será frágil y no aceptará los cambios con facilidad. Peor aún, la posibilidad de cometer un error importante en las decisiones estructurales esenciales se incrementa con un enfoque inicialmente profundo en lugar de uno amplio en primera instancia.

5.3.4. OPCIONES DE PENSAMIENTO

"*Satisfacción garantizada o le devolvemos dinero*". Esta frase es muy común a la hora de realizar la compra de algún producto. Pero cuando se trata de un software, consideraríamos que es arriesgado ofrecer garantía de satisfacción. Por lo general es muy raro que un software cuente con una garantía.
Considerando la otra perspectiva de la transacción, habría que evaluar por qué las garantías de satisfacción resultan tan atractivas. La dinámica subyacente es que las personas tienen dificultades para tomar decisiones irrevocables cuando se presenta una situación de incertidumbre. Por ejemplo, si se tiene que comprar un regalo y no se está seguro sobre las preferencias del destinatario, una garantía de satisfacción permite que se realice la compra antes de obtener la respuesta a estas interrogantes. No se obliga a tomar una decisión irrevocable hasta que se resuelva la incertidumbre, y por lo general, en condiciones de un plazo determinado, se puede devolver el producto. Si el regalo no es apropiado, lo único que se pierde es el tiempo y el esfuerzo necesarios para evaluar y devolverlo.
Sería bueno que las transacciones comerciales llevaran una cláusula de garantía de satisfacción, pero rara vez lo hacen. La mayoría de las decisiones empresariales son irrevocables y por lo general no se tiene la posibilidad de cambiar de parecer. Casi todo el mundo se resiste a tomar decisiones irrevocables en un contexto de incertidumbre. Lo óptimosería poder encontrar la manera de retrasar la toma de decisiones y proporcionar este beneficio al cliente.

5.3.4.1. RETRASAR LAS DECISIONES

Hewlett-Packard descubrió una manera de aumentar las ganancias retrasando las decisiones. HP vende una gran cantidad de impresoras en todo el mundo, y en muchos países la conexión eléctrica se debe adaptar a los tomacorrientes locales. Se pensaría que

HP podría predecir con exactitud el número de impresoras que se venden en cada país, pero los pronósticos son siempre un poco inexactos. HP siempre ha tenido algunas impresoras excedentes en un país e insuficientes en otro. Entonces a la compañía se le ocurrió la idea de hacer la configuración eléctrica definitiva en el almacén, después de que se recibe el pedido de las impresoras. Es más costoso configurar una impresora en un almacén que en la fábrica, pero en general, el costo de la opción de personalizar es compensado por la ventaja de tener siempre el producto adecuado. A pesar de que los costos unitarios aumentaron, HP ahorra $ 3 millones al mes, haciendo coincidir más eficazmente la oferta a la demanda.[21]
Enrico Zaninotto, un economista italiano, señaló que el mecanismo económico fundamental para controlar la complejidad de los sistemas "justo a tiempo" es minimizar las acciones irreversibles.[22] En el caso de HP, hubo una gran cantidad de complejidad asociada a conseguir la conexión eléctrica correcta para las impresoras que van a diferentes países. El método utilizado para controlar esta complejidad fue retrasar las decisiones sobre que conexión eléctrica instalar hasta después de recibida la orden en el almacén. Y de esta manera el sistema ya no es tan complejo.
Zaninotto sugirió que cuando un sistema que especifica con anterioridad las opciones se enfrenta a uno que mantiene las opciones abiertas, el segundo tiene la ventaja en un mercado complejo y dinámico.
Retrasar las decisiones irreversibles hasta que se reduzca la incertidumbre tiene valor económico. Conduce a realizar mejores elecciones, limita el riesgo, ayuda a controlar la complejidad, reduce los residuos y los clientes quedan satisfechos. Por otro lado, retrasar las decisiones generalmente tiene un costo. En el caso de HP, el costo unitario de la adición de un cable en el almacén era más alto que el costo de añadir el cable en la fábrica. Sin embargo, el sistema en su conjunto es más rentable, ya que el retraso permitió tomar las decisiones correctas en todo momento.

5.3.4.2. OPCIONES

Los mercados financieros y de *commodities* han desarrollado un mecanismo (llamado *opción*) para permitir que las decisiones se retrasen. Una *opción* es el derecho, pero no la obligación, de hacer algo en el futuro. Es como una garantía de satisfacción: si las cosas salen como se espera, se puede ejercer la *opción* (equivalente a mantener el producto). Si las cosas no salen bien, se puede pasar por alto la *opción* (equivalente a la devolución del producto) y todo lo que se pierde en primer lugar es el costo tal *opción*.
La incertidumbre puede moverse en dos direcciones, sin esperarlo, pueden suceder cosas buenas tan fácilmente como cosas malas. Por ejemplo, para el caso de un hotel, realizar una reserva equivale a una *opción*. El precio de la *opción* es el costo de hacer la reserva, que puede incluir un depósito anticipado. Si se ejecuta la *opción* (presentarse en el hotel), se paga el precio pactado al momento de realizar la reserva. Si se cancela el viaje se pierde la *opción*, lo que implica solamente perder el depósito realizado.
Las *opciones sobre acciones* son una manera de dar a los trabajadores la oportunidad de beneficiarse si a la empresa le va bien en el futuro y limitar el riego si a la empresa le va mal. En general, las *opciones financieras* dan al comprador la oportunidad de sacar provecho de los acontecimientos positivos en el futuro, al tiempo que limita la exposición a eventos negativos. Las *opciones* ofrecen oportunidades para tomar decisiones en el futuro, a la vez que proporcionan un seguro ante el riesgo de que las cosas salgan mal.

5.3.4.3. OPCIONES DE PENSAMIENTO PARA EL DESARROLLO DE SOFTWARE

Uno de los debates más candentes en el desarrollo de software se refiere al equilibrio entre los procesos de predicción y los procesos de adaptación. El paradigma prevaleciente ha sido el proceso predictivo: en el desarrollo de software todo debe ser especificado y detallado antes de proceder a su implementación, ya que si no cuenta con los requisitos y diseños adecuados, sin duda va a costar mucho realizar cambios más adelante. Este paradigma puede funcionar en un mundo altamente predecible. Sin embargo, si en ese mundo hay incertidumbre acerca de lo que los clientes realmente necesitan, si su situación va a cambiar, o la tecnología va avanzando, entonces un enfoque adaptativo es una buena alternativa. Las *opciones* disminuyen el riesgo de pérdidas limitando el costo y el tiempo destinado a resolver las incertidumbres. Maximizan la recompensa al retrasar las decisiones hasta obtener más conocimiento. Los economistas y gerentes de producción comprenden igualmente que el paradigma de adaptación por lo general produce mejores resultados que un enfoque predictivo.

Los procesos de desarrollo ágil se pueden considerar como la creación de opciones que permitan que las decisiones se retrasen hasta que las necesidades de los clientes estén más claramente comprendidas y las tecnologías en evolución hayan tenido tiempo para madurar. Esto no quiere decir que los métodos ágiles no son planificados, ya que los planes contribuyen a aclarar situaciones confusas, permiten el examen de las ventajas y desventajas y ayudan a establecer pautas que permitan actuar con rapidez. Por lo tanto, las planificaciones suelen incrementar la flexibilidad necesaria para responder a los cambios. Sin embargo, los planes no deberían especificar con anterioridad las acciones detalladas basadas en especulaciones. El desarrollo de software ágil persigue las especulaciones con experimentos y aprende a reducir la incertidumbre y adaptar el plan a la realidad.[23]

El conocimiento convencional en el desarrollo de software tiende a generar decisiones específicas a principio del proceso, como la congelación de las necesidades del cliente y especificar el marco teórico. En este enfoque, lo que se considera para la planificación es generalmente un proceso de predecir el futuro y tomar decisiones tempranas basándose en las predicciones sin datos o validaciones. Los planes y predicciones no son malos, pero se debe evitar la toma de decisiones irrevocables basadas en especulaciones.

En 1988 Harold Thimbleby publicó un artículo en *IEEE Software Magazine* titulado “Retrasar Compromisos” y señala que ante una nueva situación, los expertos retrasan las decisiones de la empresa mientras investigan el problema, porque saben que los compromisos que se dilatan a menudo conducen a nuevos conocimientos. Por otro lado, los amateurs pretenden tener siempre toda la razón, por lo que tienden a tomar decisiones tempranas y con frecuencia equivocadas. Una vez ejecutadas, las decisiones siguientes se basan en ellas, haciéndolas muy difíciles de cambiar. Thimbleby señala que el compromiso de diseño prematuro es un modo fallido que limita el aprendizaje, agrava el impacto de los defectos, limita la utilidad del producto e incrementa el costo del cambio.

Las *Opcionesde Pensamiento* son una herramienta importante en el desarrollo de software, siempre y cuando vayan acompañadas por el reconocimiento de que no son gratuitas y se necesita experiencia para saber cuáles mantener abiertas. Estas *opciones* no garantizan el éxito, sino que preparan las bases para el éxito si las futuras incertidumbres se mueven hacia una dirección favorable. Las *opciones* permiten decisiones respaldadas en hechos, basados en el aprendizaje más que en especulaciones.

5.3.5. EL ÚLTIMO MOMENTO RESPONSABLE

El desarrollo de software concurrente significa comenzar el proceso cuando se conocen solo requerimientos parciales y desarrollar en iteraciones cortas que proveen retroalimentación y permiten al sistema evolucionar. El desarrollo concurrente permite retrasar el compromiso hasta el *último momento responsable*,[24] es decir, el momento en el que no tomar una decisión genera la eliminación de una alternativa importante. Si los compromisos se retrasan más allá de este momento, entonces las decisiones se toman de manera predeterminada, lo cual generalmente no es un buen enfoque para la toma de decisiones.
Procrastinar no es lo mismo que decidir en el último momento responsable; de hecho, retrasar las decisiones es un trabajo duro. A continuación se presentan tácticas para la toma de decisiones en el último momento responsable:

Compartir información de diseño parcialmente completa: la idea de que el diseño debe estar completo antes de codificar es el mayor enemigo del desarrollo concurrente. Exigir información completa antes de liberar un diseño aumenta la longitud del bucle de retroalimentación en el proceso y desencadenará la toma de decisiones irreversibles mucho antes de lo necesario. Un buen diseño es un proceso de descubrimiento, realizado mediante ciclos exploratorios cortos y repetidos.
Organizar en base a la colaboración directa de trabajador a trabajador: la salida temprana de información incompleta significa que el diseño se perfeccionará a medida que avance el desarrollo. Para ello es necesario que las personas que entienden los detalles de lo que el sistema debe hacer se comuniquen directamente con la gente que entiende los detalles de cómo funciona el código.
Desarrollar un sentido de cómo absorber los cambios: anteriormente mencionamos que la diferencia entre aficionados y expertos es que estos últimos saben cómo retrasar compromisos y ocultar sus errores durante el mayor tiempo posible. Los expertos reparan sus errores antes de que causen problemas, mientras que los aficionados tratan de hacer todo bien la primera vez, sobrecargando su capacidad de resolución de problemas y tomando decisiones anticipadas erróneas. Se recomiendan algunas tácticas para retrasar el compromiso en el desarrollo de software, que podrían resumirse como un respaldo del diseño orientado a objetos y el desarrollo basado en componentes:

- Usar módulos.
- Usar interfaces.
- Usar parámetros.
- Usar abstracciones.
- Evitar la programación secuencial.
- Tener precaución con la construcción de herramientas personalizadas.

Desarrollar un sentido de lo que es de vital importancia en el dominio: olvidar alguna característica fundamental hasta que es demasiado tarde es el principal temor que impulsa el desarrollo secuencial. Las operaciones de seguridad, el tiempo de respuesta o las operaciones a prueba de fallos son de vital importancia en el dominio y son cuestiones que deben tenerse en cuenta desde el principio; si se atienden demasiado tarde estos asuntos pueden ser muy costosos. Sin embargo, la hipótesis de que el desarrollo secuencial es la

mejor manera de descubrir estos rasgos fundamentales es errónea. En la práctica, los primeros compromisos tienen más probabilidades de pasar por alto tales elementos críticos que los compromisos finales, porque los compromisos iniciales limitan el campo visual.

Desarrollar un sentido de tomar decisiones en el momento adecuado: se trata de evitar tomar decisiones por defecto o con demoras. Ciertos conceptos arquitectónicos como el diseño de usabilidad, capas y composiciones de los componentes se deben realizar al principio a fin de facilitar el surgimiento de los componentes en el resto del diseño. Una tendencia hacia compromisos tardíos no debe degenerar en una tendencia hacia ningún compromiso. Se precisa desarrollar un sentido agudo de la oportunidad y un mecanismo para lograr que se tomen decisiones cuando ha llegado su momento.

Elaborar una capacidad de respuesta rápida: Cuanto más lenta la respuesta, más adelantadas deben ser las decisiones. Si tomamos por ejemplo a la empresa Dell, ésta puede montar un ordenador en menos de una semana antes del envió y por lo tanto, puede tomar decisiones en un plazo menor a ese tiempo. A la mayoría de los otros fabricantes les toma mucho más tiempo, y es por eso que tienen que decidir qué hacer mucho antes. Si se puede cambiar un software de manera rápida, se puede esperar para hacer un cambio hasta que el cliente sepa lo que quiere.

5.3.6. TOMA DE DECISIONES

5.3.6.1. RESOLUCION DE PROBLEMAS: Enfoque en profundidad Vs. Enfoque en amplitud

Existen dos estrategias aplicables para la resolución de problemas: el enfoque de amplitud y el enfoque de profundidad. La resolución de problemas en amplitud puede imaginarse como un embudo, mientras que la resolución de problemas en profundidad es más parecida a un túnel. El primer enfoque implica compromisos demorados; mientras que el segundo significa hacer compromisos tempranos. La mayoría de las personas prefieren utilizar el enfoque de profundidad al acercarse nuevos problemas, ya que tiende a reducir rápidamente la complejidad del problema a resolver.[25] Puesto que el diseño es, por definición, la consideración de un nuevo problema, la mayoría de los diseñadores principiantes se inclinan hacia el método de profundidad.

El riesgo de la resolución de problemas con un enfoque de profundidad es que el ámbito de estudio se puede reducir demasiado pronto, sobre todo si los que toman los primeros compromisos no son expertos en el campo de estudio. Si es necesario un cambio de rumbo, el trabajo de realizar la exploración de datos se pierde, por lo que este enfoque tiene un gran costo del cambio.

Hay que tener en cuenta que los dos métodos requieren conocimientos en el dominio. El método de resolución en profundidad solo funciona si hay una selección adecuada del punto inicial del estudio. Esta correcta selección requiere dos cosas: de alguien con la experiencia necesaria para tomar las primeras decisiones correctamente y la garantía de que no habrá ningún cambio que haga obsoletas las decisiones. A falta de estas dos condiciones, el método de amplitud conducirá a mejores resultados.

El enfoque de amplitud requiere de alguien con la experiencia necesaria para entender como los detalles probablemente surgirán y la inteligencia para saber cuándo es el momento de hacer compromisos. Sin embargo el método de resolución en amplitud no

necesita un ámbito estable, ya que la técnica se basa en la selección cuando se espera que el dominio del negocio evolucione. También es un enfoque eficaz cuando el dominio es estable.

5.3.6.2. TOMAR DECISIONES INTUITIVAS

Gary Klein estudió la toma de decisiones en los servicios de emergencia, personal militar, los pilotos de líneas aéreas, personal de enfermería de cuidados intensivos y otros, para evaluar como toman decisiones de vida o muerte. Esperaba encontrar que estas personas tomaran decisiones racionales en situaciones en las que corre peligro la vida; es decir, analizar una amplia gama de opciones y evaluar los beneficios y riesgos de cada posibilidad y luego escoger la mejor opción del análisis. Cuando comenzó los estudios se sorprendió al descubrir que los comandantes de bomberos consideraron que rara vez, o nunca, toman decisiones. Los comandantes eran personas muy experimentadas y afirmaron que solo sabían qué hacer basándose en sus experiencias; no había ninguna intervención de decisiones. A esto le llamamos toma de decisiones de manera *intuitiva*.[26]
Cuando las personas con experiencia usan la combinación de modelos y la simulación mental para tomar decisiones, se está empleando una herramienta muy poderosa que tiene una historia de éxitos incuestionable. Para tomar aún mejores decisiones, los servicios de emergencia, los pilotos y los comandantes militares realizan entrenamiento situacional, que establece pautas correctas y permite mejores simulaciones mentales. Con el entrenamiento y experiencia adecuada, tomar decisiones intuitivas es un gran éxito la gran mayoría de las veces.
Klein descubrió que los comandantes de bomberos acuden a la toma de decisiones racionales cuando la experiencia es insuficiente. Deliberar acerca de las opciones es una buena idea para los principiantes que tienen que pensar en su camino a través de las decisiones. Sin embargo la toma de decisiones intuitivas es el enfoque más maduro, y por lo general conduce a mejores resultados también.[27]
La toma racional de decisiones consiste en descomponer un problema, eliminando el contexto, la aplicación de técnicas analíticas y exponer el proceso y los resultados de la discusión. Este tipo de toma de decisiones genera mejoras incrementales, pero padece de una “visión de túnel”, ignorando deliberadamente los instintos de las personas con experiencia. Ayuda a aclarar situaciones complicadas, pero contiene ambigüedad significativa. A pesar de que el análisis racional da respuestas concretas, éstas se basan en suposiciones difusas y es difícil saber exactamente cuándo y cómo aplicar las reglas.
Sería bueno que el análisis racional se pudiera expresar de manera explícita para señalar cuando hay una incoherencia o un factor clave que todo el mundo está pasando por alto. Sin embargo, el análisis racional es menos útil en este sentido, porque tiende a eliminar el contexto de análisis. Por lo tanto, la toma racional de decisiones difícilmente podrá detectar errores de alto riesgo, siendo la toma intuitiva de decisiones mejor en este sentido.

5.3.6.3. REGLAS SENCILLAS PARA EL DESARROLLO DE SOFTWARE

Las reglas simples dan a las personas un marco para la toma de decisiones en vez de ser un conjunto de instrucciones que indican exactamente lo que se debe hacer. Por lo tanto, las reglas simples son principios que se aplicarán de manera diferente en ámbitos diferentes.

Estas son utilizadas por las personas experimentadas como guía al tomar decisiones intuitivas.
Las personas tienen un número límite de cosas que pueden tener en cuenta al tomar una decisión, por lo que la lista de reglas debe ser corta. Debe limitarse a pocos principios clave que realmente deban ser considerados al tomar una decisión. Muy a menudo se utilizan reglas simples para reforzar un cambio de paradigma, centrándose en los elementos contrarios a la intuición en la toma de decisiones.[28]
Lean Software Development ofrece siete reglas sencillas (o principios) para el desarrollo de software:

1. **Eliminar residuos**: utilizar el tiempo solamente en lo que suma valor real al cliente.
2. **Amplificar el aprendizaje**: cuando surgen problemas difíciles, aumentar la retroalimentación.
3. **Decidir lo más tarde posible**: mantener las opciones abiertas mientras sea posible, sin excederse.
4. **Entregar lo más rápido posible**: entregar valor al cliente tan pronto como lo pidan.
5. **Facultar al equipo**: que las personas que agregan valor utilicen todo su potencial.
6. **Construir la integridad**: no tratar de trabajar en integridad sin antes construirla.
7. **Ver el todo**: no caer en la tentación de optimizar partes a expensas del conjunto.

Un conjunto de reglas sencillas son hitos; su propósito es permitir a las personas tomar decisiones rápidas sobre cómo proceder, sabiendo que sus decisiones no serán arbitrarias, porque estarán tomando la misma opción que sus directivos tomarían en las mismas circunstancias. Es el poder que las reglas simples dan a las personas lo que las hace tan valiosas. No es importante que las reglas ofrezcan una guía detallada; sino que son pautas, lo que da a las personas la libertad de tomar sus propias decisiones.

5.4. CUARTO PRINCIPIO - ENTREGAR LO MAS RAPIDO POSIBLE

5.4.1. "LA PRISA GENERA DESPERDICIO"

Cuando se habla que entregar lo más rápido posible, no se trata de correr a una velocidad vertiginosa para cumplir con los objetivos. La entrega rápida es una práctica operativa que proporciona una fuerte ventaja competitiva. A los clientes les gusta tanto la entrega rápida que una vez que una empresa en una industria aprende cómo entregar más rápidamente, generalmente se espera que la competencia imite ese concepto.
A finales de 1980, a las compañías japonesas les llevó un promedio de 46 meses, 485 trabajadores y 1,7 millones de horas de mano de obra para terminar un nuevo automóvil para el mercado. Los fabricantes de automóviles de Estados Unidos tuvieron un promedio de 60 meses, 903 trabajadores y 3,1 millones de horas de labor.[29] Al hacer referencia sobre estos números, James Womack dijo: "Sugerimos que a la frase 'más rápido es más caro' debe sumarse la frase 'la calidad cuesta más' en la pila desperdicios sobrantes de la era de la producción en masa".[30]

5.4.2. ¿QUE SIGNIFICA ENTREGAR LO MAS RAPIDO POSIBLE?

Los clientes prefieren la entrega rápida. Es por eso que el envío inmediato y la entrega rápida es la metodología estándar para los catálogos online y de pedidos por mail. La entrega rápida permite a los clientes demorar o retrasar las decisiones, y para otros clientes significa satisfacción más prontamente. Para los clientes de desarrollo de software, esto a menudo se traduce en una mayor flexibilidad empresarial.
Mientras los clientes están descubriendo los beneficios de la entrega rápida, las empresas que lo aplican están ahorrando dinero. Entrega rápida significa que las empresas pueden entregar el producto antes que los clientes puedan cambiar de opinión. Esto significa que las empresas tienen menos recursos inmovilizados en los procesos, ya sea en stock o parcialmente realizado. Cuando el trabajo en proceso representa un riesgo, la entrega rápida reduce el peligro.
Por ejemplo, Dell Computer considera que la obsolescencia del stock es su mayor riesgo; por lo tanto espera hasta recibir una orden de pedido y entonces fabrica y envía las maquinas en menos de una semana. De éste modo, cuando una tarjeta de vídeo rápida o una unidad de disco más grande están disponibles, Dell puede ofrecer la parte mejorada de manera más rápida de lo que sus competidores puedan hacerlo. Una vez que la mejora se ofrece en las maquinas Dell, sus competidores a menudo se encuentran en la situación de tener que ofrecer también una nueva pieza, suprimiendo una cantidad importante de stock.
Una gran pila de trabajo en proceso conlleva riesgos adicionales además de la obsolescencia. Los problemas y defectos, tanto grandes como pequeños, a menudo se ocultan en las pilas de trabajo parcialmente realizado. Cuando los desarrolladores crean una cantidad grande de código sin realizar las pruebas, peligra la acumulación de defectos. Cuando el código es desarrollado pero no integrado, la parte del esfuerzo que conlleva mayor riesgo usualmente se mantiene. Cuando un sistema está completo, pero no en producción, los riegos siguen permaneciendo. Todos estos riesgos pueden reducirse significativamente mediante el acortamiento de la cadena de valor.

El principio de entregar lo más rápido posible se complementa con decidir lo más tarde posible. Mientras más rápido se realiza la entrega más tiempo se pueden retrasar las decisiones. Por ejemplo, si se tiene que hacer un cambio de software en una semana, entonces no se tiene que decidir exactamente lo que se va hacer hasta una semana antes de que el cambio se lleve a cabo. Por otro lado, si lleva un mes realizar cambios, entonces se tiene que decidir sobre los detalles del cambio de todo un mes antes de que se cumpla el plazo de entrega. La entrega rápida es un enfoque de opciones amigables para el desarrollo de software. Esto permite mantener las opciones abiertas hasta que se haya reducido la incertidumbre y se pueda tomar decisiones más fundamentadas y basadas en hechos.

5.4.3. SISTEMAS PULL

La entrega rápida no surge por accidente. Cuando las personas se presentan a trabajar, deben encontrar la manera de invertir su tiempo. Debe quedar claro para cada persona, en todo momento, qué debe hacer para contribuir de manera más eficaz a la empresa. Cuando las personas no conocen qué deben realizar, se produce pérdida de tiempo, la productividad se resiente y la entrega rápida no se hace posible.
Hay dos maneras de asegurar que los trabajadores hacen el uso más efectivo de su tiempo. Una es instruirles sobre qué deben hacer y la otra es disponer de las tareas para que ellos puedan darse cuenta por sí mismos de como es el trabajo a llevar a cabo. En un ambiente de trabajo con rápido movimiento, sólo la segunda opción funciona. Las personas que se ocupan habitualmente de situaciones fluidas, como por ejemplo los trabajadores de emergencias y el personal militar, no dependen de un director a distancia para decirles cómo responder a los acontecimientos. Estas personas descubren cómo reaccionar frente a eventos junto con las otras personas que también se encuentran en la escena.
Cuando las cosas transcurren con rapidez, no hay suficiente tiempo para que la información viaje por la cadena de mando y luego vuelva en forma de directivas. Por lo tanto, los métodos para señalización local y compromisos deben ser desarrollados para coordinar el trabajo. Una de las maneras claves para hacer esto es dejar que las necesidades del cliente "*tiren"* (pull) del trabajo en lugar de tener un programa o un cronograma que "*empuje"* (push) para que se realice el trabajo.

5.4.3.1. CRONOGRAMAS DE MANUFACTURA

En las plantas de fabricación complejas, uno de los desafíos más grandes es averiguar exactamente lo que cada máquina y cada persona debe estar haciendo en cualquier momento con el fin de maximizar el rendimiento. En la década de 1980, hubo un esfuerzo conjunto para usar software MRP*(Planificación de Requerimientos de Materiales)* para programar las tareas en el taller. MRP es básicamente una herramienta de planificación, por lo que la idea fue: además de los materiales de planificación, ¿por qué no planificar la producción también?
Habiendo sido ineficientes en la planificación de materiales, los sistemas MRP fueron un desastre cuando se utilizaron para la planificación en una planta de producción. Esto se debía al nerviosismo que se generaba al introducir el menor cambio, ya que el nuevo plan para lidiar con el problema era completamente diferente al último plan. Después de re planificar nuevamente, todo lo que se venía realizando en la planta se debía detener y hacer otra cosa. El viejo cronograma había sido el mejor esquema basado en las en antiguas

suposiciones, mientras que el nuevo cronograma fue optimizado basado en nuevos supuestos. El hecho de que esto cambió lo que cada persona y maquina en la planta estaba haciendo no hace ninguna diferencia en el equipo.
El simple hecho matemático que se aplica aquí es que la variación siempre se amplifica a medida que avanza por una cadena de acontecimientos conectados. Una pequeña variación en el primer paso genera una enorme variación cinco pasos más adelante. Muy a menudo, los operarios de producción eran culpados por no hacer exactamente lo que estaba previsto, pero ese no era precisamente el problema. El problema era que cuando surgía incluso la falla más pequeña, el cronograma se invalidaba, y a partir de entonces, si se lo seguía se terminaban haciendo peor las cosas.
El método Justo a Tiempo modificó todo esto introduciendo en la fabricación el concepto de *planificación de"demanda"* (Pull). Los Sistemas Pull utilizan un mecanismo llamado *Kanban*, que significa señal o cartel en japonés.[31] A continuación de detalla el funcionamiento de un sistema *Kanban*:

Cuando se recibe un pedido en un sistema kanban, la orden se envía inmediatamente a una estación de despacho y los trabajadores del área buscan en sus estantes las piezas que necesitan para completar el pedido. Cada parte tiene adjunta una tarjeta de identificación Kanban; la pieza se retira y se deja la tarjeta en el estante. Luego alguien lleva la tarjeta al área de suministros que se encarga de hacer ese tipo de partes para su reposición. Una vez confeccionada, se adjunta la tarjeta y la pieza se repone en el estante de la estación de despacho. A su vez, el área de suministros tiene sus propios estantes y sus proveedores para mantener su stock. Este flujo se mantiene a través de la planta, con todo el trabajo programado por las tarjetas Kanban.

Kanban constituye el mecanismo propicio para el método Justo a Tiempo. Es lo que le dice a las personas y máquinas que hacer de una hora a otra, a fin de lograr un rendimiento óptimo de la planta. A diferencia de otros mecanismos de planificación, los Sistemas Pull toman en cuenta la variabilidad al final de la línea de trabajo, por lo que se genera muy poco nerviosismo.
Sin la planificación Pull y las tarjetas Kanban, tendría que haber alguna otra manera para que los operarios puedan averiguar que tarea realizar. De hecho, en épocas anteriores a Lean, las personas recibían instrucciones de los directores, que modificaban el calendario de Planificación de Requerimiento de Materiales (MRP) en base a su conocimiento personal y decidían lo que cada estación de trabajo debía hacer. Se puede imaginar que se trataba de una cuestión de prueba y error en una planta compleja. Lo interesante de la programación Pull es que se tiene al gerente fuera del circuito de tener que decirles a los trabajadores lo que tienen que hacer. El trabajo es auto-dirigido y de esta manera los gerentes utilizan su tiempo para entrenar al equipo.[32]

5.4.3.2. PLANIFICACON EN EL DESARROLLO DE SOFTWARE

En el ámbito del desarrollo de software complejo, existe el mismo problema de fondo: ¿Cómo asegurarse de que los trabajadores sepan cómo utilizar su tiempo de la forma más efectiva para lograr el objetivo en cuestión? A falta de un mejor método para que los desarrolladores resuelvan esto por sí mismos, los directores con frecuencia consultan el

calendario de planificación del proyecto, lo modifican en caso de ser necesario en función a sus conocimientos personales y comunican a los desarrolladores lo que deben hacer.
Pero el problema es que un cronograma de proyecto será tan confiable como un cronograma MRP o un plan maestro de producción si es utilizado para la planificación detallada en un ámbito que experimenta incluso una pequeña cantidad de variabilidad. Además, decirles a los desarrolladores que hacer no genera mucha motivación.[33]
A menudo se escuchan quejas sobre la micro gestión en el desarrollo de software. Es comprensible por qué los gerentes pueden sentir la necesidad de proporcionar instrucciones detalladas a los desarrolladores si el trabajo no está organizado para ser auto-dirigido y no existen mecanismos de señalización y compromiso. Si un sistema es complejo, los recursos son escasos y los plazos ajustados, entonces todo el mundo debe ser productivo todo el tiempo. La pregunta sería: ¿De qué manera las personas van a saber cómo usar mejor su tiempo a menos que alguien les diga qué hacer?
No usar un cronograma puede permitir hacer asignaciones eficaces de trabajo detallado en un entorno complejo con incluso una variabilidad modesta. Depender de cronogramas computarizados para hacer asignaciones de trabajo y decirles a los desarrolladores lo que deben hacer no son las mejores maneras de manejar situaciones complejas o cambiantes. Un enfoque más eficaz es utilizar Sistemas Pull que crean mecanismos de señalización y de compromisos adecuados, de manera que los miembros del equipo puedan averiguar por si mismos la forma más productiva de invertir su tiempo.

5.4.3.3. SISTEMAS DE SOFTWARE PULL

El punto de partida de un Sistema Pull en el desarrollo de software son las iteraciones cortas basadas en la información brindada por el cliente al inicio de cada bucle.[34] Supongamos que en el comienzo de cada iteración, los clientes o sus representantes escriben en tarjetas descripciones de las características que necesitan.[35] Hay muchas otras maneras de documentar lo que quieren los clientes, pero las tarjetas índices se parecen mucho a las tarjetas Kanban, por lo tanto por el momento se asumirán como tales.
Como se describe en el Capítulo 2, "Amplificar el aprendizaje", los desarrolladores estiman cuánto tiempo tomará implementar cada tarjeta y los clientes dan prioridades a las tarjetas. Al finalizar la reunión de planificación, el trabajo para la iteración está contenido en las tarjetas seleccionadas para su implementación. Estas tarjetas se convierten ahora en tarjetas Kanban, que básicamente le dicen al equipo de desarrollo el trabajo que deben hacer durante el transcurso de la iteración.
Recordemos que la idea es generar trabajo auto-dirigido. Por lo tanto, las tarjetas no son asignadas a los desarrolladores, sino que ellos eligen con que tarjetas quieren trabajar. Las tarjetas se pueden publicar en el área de *"Tareas a Realizar"* de una pizarra, donde los desarrolladores van a averiguar qué hacer. Luego las tarjetas en las que se trabaja se mueven a otra área de la pizarra de *"Tareas Chequeadas"*. Una vez que se pasan las pruebas, la tarjeta se mueve al área de "*Aprobadas"*.
Las tarjetas por si solas no son suficientes para qué los desarrolladores conozcan exactamente qué hacer. Una breve reunión regular, preferentemente diaria, es también una buena idea para ayudar hacer el trabajo de la iteración auto-dirigida. Las reuniones del equipo no deben durar más de 15 minutos diarios, y debe ser realmente reuniones de trabajo. Todo el equipo debería estar presente, incluso si eso significa llamados telefónicos; y la participación activa en general debe limitarse a los miembros del equipo.

En las reuniones diarias, los miembros del equipo dan un resumen de lo que realizaron el día anterior, lo que se va a realizar en el día de la fecha, y en qué necesitan ayuda. Si algunos asuntos requieren discusiones más detallados, éstos se tratan en reuniones posteriores de las partes interesadas. El trabajo del líder proporciona la interferencia en el equipo. Por ejemplo, si un desarrollador necesita más información de los clientes, el trabajo del líder es asegurarse de que el desarrollador tenga acceso a los clientes o sus representantes para responder a las preguntas.[36]
En el desarrollo de software, un sistema pull requiere un tiempo de planeamiento corto (un mes o menos), de lo contrario puede transformarse en un sistema push. Si la iteración es muy larga, habrá demasiadas tarjetas o bien las tarjetas no serán lo suficientemente detalladas como para que el sistema pull trabaje efectivamente. Un Sistema Pull trabaja a partir de los pedidos del cliente y usa múltiples mecanismos de señalización y de compromiso para organizar el trabajo de manera auto-dirigida.

5.4.3.4. RADIADORES DE INFORMACION

Una de las características de un Sistema Pull es el control visual o de administración por vista.[37] Si el trabajo será auto-dirigido, entonces todo el personal debe ser capaz de ver lo que está sucediendo, lo que se debe hacer, qué problemas existen y qué progresos se están realizando. El trabajo no puede ser auto-dirigido hasta que los controles visuales sean los apropiados para el dominio, actualizados y utilizados para dirigir el trabajo. AlistairCockburn denomina a los controles visuales para el desarrollo de software como *radiadores de información*.[38] La tarjeta Kanban es un radiador de información que muestra varias cosas: lo qué hay que hacer, lo que ya está hecho, y quién está trabajando en qué.
Las listas de problemas, ideas para mejoras, candidatos para refactorización, impacto del sistema a la fecha, status diarios de desarrollo, glosarios con los lenguajes comunes utilizados y bancos de prueba de base de datos, son todos candidatos para ingresar a una tabla grande y visible. Los radiadores de información hacen los problemas visibles y constituyen un mecanismo que facilita el trabajo auto-dirigido.

5.4.4. TEORIA DE COLAS

Con frecuencia se dice: “Mi mayor problema es el departamento de pruebas”. Si bien las personas del departamento de pruebas son dedicadas, trabajadoras e importantes para el esfuerzo del desarrollo, parecería que nunca hay suficientes personas de éste tipo; y aunque los desarrolladores hacen sus propias pruebas de unidades, los testers con frecuencia hacen las pruebas de aceptación.[39] Por lo tanto, sin suficientes testers, todo el proceso de desarrollo se atasca.
El cuello de botella puede no ser los testers, podrían ser los analistas o es posible que haya problemas para obtener información de los clientes. O quizás solo exista un responsable que comprende o entiende la base de datos heredada. Sea cual sea el cuello de botella, una breve mirada a la teoría de colas podría ayudar al abordaje del problema.

5.4.4.1. REDUCCION DEL TIEMPO DE CICLO

Cualquier tipo de cola representa un gasto de tiempo, ya sea en el tráfico, esperando en una tienda, al teléfono, etc. La teoría de Colas se ocupa de hacer la espera lo más corta posible, brindando mejoras en todos los ámbitos en la que ha sido aplicada. Probablemente, incluso se utiliza para calcular el número de servidores que se debe tener en una sala de ordenadores.
La medida fundamental de una cola es el *tiempo de ciclo*, es decir, el tiempo promedio que se requiere para ir de un extremo al otro de un proceso. El tiempo de ciclo inicia cuando algo entra en una cola y sigue corriendo mientras se espera en la misma a medida que se efectúa el servicio, y así sucesivamente hasta que se sale en el otro extremo del proceso.
Hay que tener en cuenta que cuando se está en una cola, siempre se busca que el tiempo de ciclo que sea lo más corto posible. Ya que después de todo, se ingresa a una cola para lograr algo. La única razón por la que no se puede lograr el objetivo inmediatamente es que los recursos necesarios sean limitados. El tiempo dedicado a la espera en la cola es tiempo perdido.

5.4.4.2. TASA DE LLEGADA CONSTANTE

Hay dos maneras de reducir el tiempo de ciclo; una se basa en la forma en que llega el trabajo y la otra en la manera en que se procesa. En algunos sistemas, no es posible influir en la tasa de llegada de trabajo, pero en otros se puede establecer políticas para equilibrar la demanda entrante. Las políticas de precios a menudo se utilizan para este propósito. Una compañía telefónica que ofrece tarifas muy bajas por la noche y durante el fin de semana hace esto para nivelar la demanda máxima. Los consultorios médicos utilizan sistemas de reserva para asegurar que los pacientes lleguen a intervalos regulares. Cuando el ingreso de demanda se extiende para igualar la capacidad del sistema, las colas, y por lo tanto el tiempo de ciclo, serán reducidos.
Una forma de controlar la tasa de llegada de trabajo es lanzar pequeños paquetes de trabajo. Si se debe esperar que un gran lote de trabajo llegue antes de que pueda comenzar a procesarlo, entonces la cola será tan larga como el lote completo. Si el mismo trabajo se libera en pequeñas cantidades, la cola puede ser más pequeña.
Las organizaciones de desarrollo de software a menudo controlan la llegada de trabajo con un proceso de revisión que establece las prioridades y selecciona los proyectos. Si se trata de un evento anual ligado a un proceso presupuestario, entonces el valor del trabajo de un año llega de una sola vez, generando muy largas colas. Incluso con un proceso de aprobación de proyectos trimestrales, las colas son aún muy grandes. Muchos gerentes creen que es bueno para los proyectos grupales en un único proceso de establecimiento de prioridades, tener más proyectos para comparar a la vez. La teoría de colas sugiere que probablemente sería mejor liberar los proyectos con mayor frecuencia, mensual (o incluso semanal), para nivelar la llegada de trabajo en el área de desarrollo.

5.4.4.3. TASA DE SERVICIO CONSTANTE

Una vez que se ha eliminado la variabilidad en la llegada de trabajo en una cola, el siguiente paso es eliminar la variabilidad en el tiempo de proceso. Los paquetes pequeños de trabajo son de gran ayuda en la eliminación de la variabilidad en el tiempo de procesamiento, ya

que estos tienen menos cosas que pueden fallar. Sin embargo, incluso con los paquetes pequeños de trabajo, puede ser difícil determinar cuánto tiempo consumirá cada uno. La forma más sencilla de resolver este problema es aumentar el número de servidores que procesen el trabajo en una sola cola. Los bancos y mostradores de los aeropuertos no tienen una manera fácil para determinar qué clientes tomarán una gran cantidad de tiempo, por lo que reducen la variabilidad al tener una única cola alimentando varias estaciones. Algunas de las estaciones pueden quedar paralizadas con los clientes que permanecen demasiado tiempo, pero la línea principal todavía puede ser atendida a un ritmo constante por las estaciones restantes.
De esta manera vemos la importancia de los paquetes pequeños de trabajo. No sólo estos fluyen más fácilmente a través de su sistema, sino que también permitirán el procesamiento paralelo de los pequeños trabajos de varios equipos, de manera que si uno está estancado por un problema, el resto del proyecto puede llevarse a cabo sin demora.
Si se tiene un proceso que implica varios pasos o etapas, entonces el tiempo de procesamiento en los pasos iniciales afectará la velocidad a la que el trabajo llega a las estaciones posteriores. Si se tiene grandes variaciones de procesamiento en los puestos de trabajo principales, estas repercutirán en todo el sistema. Por lo tanto, es buena idea tratar de trasladar cualquier variabilidad desde el principio.
Es muy importante entender el serio impacto de las variaciones cuando se utiliza desarrollo iterativo. Digamos que se tiene un cuello de botella en las pruebas de aceptación, y si esto es lo último que sucede antes de la implementación, entonces el cuello de botella no afectará sensiblemente el trabajo inicial. Pero cuando se realizan iteraciones, las pruebas de aceptación no son el último paso, sino que son una parte vital de cada iteración y deben realizarse antes de proceder a la siguiente. Si se salta éste paso vital, no se obtendrá retroalimentación, que es un objetivo clave de la iteración en primer lugar. Con el desarrollo iterativo, las pruebas de aceptación repercuten a lo largo de todo el proceso, y cualquier retraso se amplificará en iteraciones posteriores. Por lo tanto es muy importante no tener un cuello de botella en la etapa de pruebas.

5.4.4.4. ESTANCAMIENTO

La manera más evidente de reducir el tiempo del ciclo es tener gran capacidad para procesar el trabajo. No es posible tener tiempos de ciclos cortos si se sobrecargan los recursos. Los gerentes de operaciones saben que a medida que se acerca la utilización completa de los servidores, el tiempo de ciclo para el procesamiento de las solicitudes a ese servidor se alarga de forma dramática.
Consideremos que el departamento de pruebas se ha convertido en un cuello de botella. Si no se hace caso a la necesidad de tener más personas, el director intentará hacer el mejor uso del personal disponible, asegurándose de que un lote de trabajo de gran tamaño siempre este esperando en la cola para que el personal esté ocupado. Los desarrolladores mantienen la codificación a la espera de los resultados de las pruebas, debido a que su gerente también quiere mantener al personal ocupado. A pesar de que es más barato corregir un error inmediatamente después de que se codifica en lugar de hacerlo en una o dos semanas más tarde, este sistema anima a los desarrolladores a crear cada vez lotes de código más grandes para ser probados. La cola de las pruebas crece y el trabajo de desacelera. El personal comienza a probar menos minuciosamente, los errores se liberan, y la cola sigue creciendo. Utilizando en su totalidad la capacidad de pruebas, el costo para la

empresa puede ser significativo. Este es un caso en el que los administradores no entienden la teoría de colas y crean políticas que en realidad tienen el efecto contrario a lo que se pretende.
Otra política autodestructiva es retrasar las pruebas de aceptación hasta que toda la codificación se termina y la unidad es probada. De nuevo, ésta política asegura que el trabajo llegue al departamento de pruebas en grandes lotes.

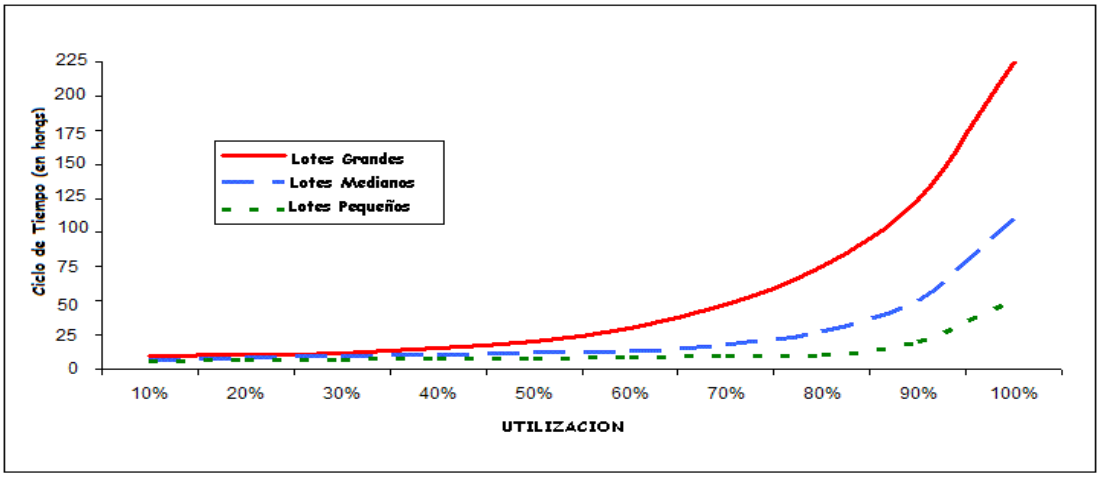


Fig. 5.8 Efecto de la utilización y tamaño de lotes en el tiempo de ciclo.

Supongamos que cada lote grande puede ser probado en 7 horas si nada más se encuentra en la cola. El gráfico en la Figura 8.1, muestra que cuando el departamento de pruebas trabaja al 50 por ciento de su capacidad, el tiempo necesario para realizar la prueba del lote será alrededor de 25 horas, y al 85 por ciento de su capacidad el tiempo del ciclo es superior a 100 horas y se incrementa rápidamente. Esto funciona igual que un embotellamiento de tráfico en hora pico, al 85 por ciento de la capacidad el estancamiento es inevitable.
Ahora, asumamos que el departamento puede ser persuadido de que mover las pruebas rápidamente es el mejor enfoque. Entonces se elimina la política de “solo trabajar con sistemas completos” y se aceptan características para ser probadas tan pronto como son codificadas.
Consideremos que los lotes pequeños pueden moverse a través del departamento en 4 horas a niveles bajos de utilización. Debido a que los lotes son pequeños, el departamento debería poder mantener un tiempo de respuesta de 5 horas y utilizando un 70 por ciento de su capacidad, sin presentarse demoras hasta alcanzar el 90 por ciento, momento en el cual los trabajos pequeños se mueven a través del departamento ocho veces más rápido que los trabajos grandes. Esto da como resultado que el departamento procesa los trabajos mucho más rápido, a una mayor capacidad.
Muchas firmas consultoras utilizan *proporción aplicada* como una medida de gestión clave y que consideran que se debe maximizar, ya que afecta directamente a la utilización de los beneficios. Medidas similares han encontrado su camino dentro de las organizaciones de software, en las que el vínculo con la rentabilidad es más tenue. Es difícil para aquellos que piensan de esta manera entender que la completa aplicación no proporciona ningún valor general a la cadena de valor, de hecho, normalmente produce más daños que beneficios.
En su libro *“Slack”*, Tom DeMarco hace notar que tener holgura en una organización le da la capacidad de cambiar, de reinventarse a sí misma y reunir recursos para el crecimiento. En

realidad, la teoría de colas sugiere que la holgura sirve para un propósito aún más básico. Al igual que una carretera no puede proporcionar un servicio aceptable sin una cierta holgura en la capacidad, probablemente no se proporcionará a los clientes el más alto nivel de servicio si no se cuenta con holgura en la organización.

5.4.4.5. TEORIA DE RESTRICCIONES

Según la Teoría de las Restricciones,[40] la mejor manera de optimizar una organización es centrarse en el rendimiento de la misma, ya que ésta es la clave para generar ingresos. La forma de aumentar el rendimiento es buscar el cuello de botella actual que está frenando las cosas, solucionarlo y así sucesivamente. Manteniendo esto, se tendrá una cadena de valor de movimiento rápido.

Se debe notar que no sirve de nada aumentar la utilización de las áreas que no producen cuello de botella. No importa que tan rápido se desarrolle software si no puede ser probado a la misma velocidad. No importa la rapidez con que se desarrolle un sistema si no se tiene el personal para implementarlo. Entonces se debe mover personal hacia el cuello de botella sin seguir acumulando trabajo que no se puede usar inmediatamente.

5.4.4.6. FUNCIONAMIENTO DE LAS COLAS

La Teoría de Colas es una disciplina bien conocida cuando algo fluye a través de un recurso limitado. A continuación se resume cómo funcionan las Colas:

- La medición de la cantidad de trabajo que espera ser realizado (Trabajo en Cola) es equivalente a la medición del tiempo de ciclo de un sistema.[41]
- A medida que la variabilidad aumenta (en la hora de llegada o el tiempo de procesamiento), el tiempo de ciclo y la cola de trabajo se incrementará.
- Al aumentar el tamaño del lote, la variabilidad en los arribos y el tiempo de procesamiento aumentan, y por lo tanto el tiempo del ciclo y los trabajos en colas se incrementarán.
- A medida que la utilización aumenta, el tiempo del ciclo se incrementará de forma no lineal.
- Al aumentar la variabilidad, el incremento no lineal en el tiempo del ciclo ocurre en niveles de utilización cada vez más bajos.
- Un flujo continuo requiere de reducción en la variabilidad.
- La variabilidad se puede reducir mediante un arribo equilibrado de la demanda, lotes pequeños, una tasa de procesamiento equilibrada y procesamiento en paralelo.
- La disminución temprana de la variabilidad en el proceso tiene un impacto más grande que la disminución tardía de la variabilidad en el proceso.

Los gerentes de Desarrollo de Software tienden a ignorar los tiempos de ciclo, tal vez porque suponen que ya están realizando las cosas tan más rápido como pueden. De hecho, la reducción del tamaño de los lotes y hacer frente a los cuellos de botellas puede reducir un poco el tiempo del ciclo, incluso en organizaciones que se consideran eficientes.

5.4.5. COSTO DEL RETRASO

El desarrollo de software es un proceso de descubrimiento en el que los técnicos continuamente toman las decisiones de equilibrio con el fin de llegar a lo que ellos consideran un resultado óptimo. Desde luego, los técnicos traen sus perspectivas únicas a su trabajo, por lo que sus decisiones serán influenciadas por sus antecedentes y experiencia. Uno de los mayores retos para los líderes de desarrollo de software es asegurar que las constaste decisiones de equilibrio tomadas por todo el equipo produzcan un resultado óptimo.
Muy a menudo, a un equipo de desarrollo de software se le dice que debe cumplir simultáneamente con el costo, características y objetivos de fecha de presentación; no puede haber concesiones. Esto envía dos mensajes al equipo:

- Los costos de soporte no son importantes, debido a que no fueron mencionados.
- Cuando algo tiene que ceder, se debe realizar un análisis personal de las ventajas y desventajas.

Dado que varios miembros del equipo pueden tener distintas percepciones de lo que es importante, las acciones hechas por algunos probablemente serán contrarrestadas por acciones realizadas por otros, comprometiendo todos los objetivos.
Se debe dar al equipo un modelo económico y facultar a los miembros para poder averiguar por sí mismos lo que es importante para el negocio. Con esto se les da a todos el mismo marco de referencia para que puedan trabajar a partir de los mismos supuestos. Finalmente, el equipo tiene más posibilidades de llegar a un resultado económico positivo, ya que ahora saben lo que significa el éxito económico.

5.4.5.1. MODELO DE APLICACIÓN

Si una organización de desarrollo de software no interviene en el desarrollo de productos, es útil desarrollar un modelo económico de cada aplicación desde el punto de vista del cliente. Esta es una manera simplificada de evaluación de cómo las diferentes decisiones de diseño afectan al valor de negocio recibido por el cliente. Una simple mirada al modelo económico del cliente ayuda al equipo a tomar las decisiones de equilibrio de la aplicación.
El primer paso en el desarrollo de un modelo de aplicación es identificar los factores económicos de sus clientes relacionados con la aplicación. Si se está trabajando con una empresa que no puede proveer información financiera detallada, incluso algunas estimaciones aproximadas serían útiles. Para la realización del modelo de aplicación será esencial la ayuda del contador de la empresa.

5.4.5.2. DECISIONES DE EQUILIBRIO

La razón para desarrollar modelos económicos simples de un proyecto de desarrollo es proporcionar al equipo una guía para la toma de decisiones de equilibrio.
Los modelos económicos han sido utilizados al momento de decidir qué proyectos financiar, pero su uso en la toma de decisiones durante el desarrollo ha sido limitado. Se podría decir que basar las decisiones de desarrollo en modelos económicos ayuda a que el equipo tome buenas decisiones de equilibrio. Proporciona a las personas lineamientos para realizar

concesiones que conducen a tomar decisiones más eficaces. Esto permitirá a los desarrolladores que se sienten capaces y a una organización responder y prosperar en un entorno competitivo.
Por último, los modelos económicos pueden ayudar a justificar el costo de reducir el tiempo del ciclo, eliminando los cuellos de botella y adquiriendo herramientas que permitirán entregar lo más rápido posible el producto.

5.5. QUINTO PRINCIPIO - POTENCIAR AL EQUIPO

5.5.1. LA ADMINISTRACION CIENTIFICA

Cuando Henry Ford introdujo el Modelo T en 1908, tuvo tanto éxito que la compañía tuvo que inventar continuamente formas mejores y más rápidas para la fabricar un automóvil. La primera línea de montaje móvil se introdujo en 1913, y hacia el año 1927, la planta de River Rouge en Dearborn, Michigan, transformaba hierro en automóviles terminados en tan sólo 28 horas. Ford produjo casi 17 millones unidades del Modelo T, transformando al país en sólo dos décadas.
Con la línea de montaje de Ford comenzó la era del ingeniero industrial y del supervisor, dirigiendo un equipo de trabajo y diciendo cómo debían hacer su trabajo. Al principio, el sueldo eran muy bueno, pero después de un tiempo los trabajadores comenzaron a darse cuenta de que estaban atrapados en puestos de trabajo degradantes. Por este motivo los sindicatos se hicieron fuertes y un manto de insatisfacción se asentó en la industria.
Es interesante observar que los reportes evaluativos empezaron a surgir en los métodos de trabajo por la década de 1910, justo en el momento en que la gestión científica fue ganando credibilidad en la producción industrial. Con el tiempo, las evaluaciones de desempeño se convirtieron en una especie de boletines de calificaciones de la industria. Durante décadas, se daba por hecho que la remuneración era el motivador más eficaz para los trabajadores, y no fue sino hasta la década de 1970 que esta hipótesis empezó a ser cuestionada.[42]
En la década de 1980, se hizo evidente que las técnicas de fabricación impulsadas por Toyota, más tarde llamadas *Manufactura Lean*, podían producir productos de alta calidad más rápido y más económicamente que las técnicas de Gestión Científica. Con sus teorías motivacionales y operativas en entredicho, los gerentes comenzaron a avanzar más allá de la Gestión Científica.
Así se iniciaron una serie de programas con nombres tales como MBO, TQM, Zero Defects, OptimizedOperations, ServiceExcellence, ISO9000, Total ImprovementProgram y CustomersFirst (Atención al Cliente), todas destinadas a enriquecer el ambiente de trabajo y el resultado final. En ocasiones con éxito, la gran variedad de programas generalmente produjeron resultados insignificantes. Con demasiada frecuencia, estos programas no han cambiado la realidad de cómo se realizó el trabajo. En muchos casos, aumentaron la intensidad de los factores que conducen a la insatisfacción en el trabajo (política, supervisión, administración) en lugar de aumentar los factores que contribuyen a la satisfacción laboral (logro, reconocimiento, responsabilidad).[43] Si bien esto no es culpa del programa, es un efecto secundario común.

5.5.2. CMM

El programa de mejora de desarrollo de software más conocido es el *Modelo de Madurez de Capacidades* o *CMM* (CapabilityMaturityModel). Al igual que con otros programas, CMM ha tenido una serie de resultados que van desde éxitos espectaculares a decepciones. Ha sido utilizado como un programa de certificación similar a ISO9000, sobre todo por empresas de desarrollo de software que buscan hacer negocios en otros países. En la medida en que se ha utilizado como un programa de certificación, CMM ha tenido un impacto similar al

ISO9000, tendiendo ambos a crear burocracia y hacer difíciles los cambios, pese a que no es la intención de ninguno de los dos programas.
Los programas ISO9000 no deben ser considerados como programas de mejora de procesos, ya que tienen tendencia a documentar y por lo tanto estandarizar los procesos existentes en lugar de mejorarlos. Dado que los programas ISO9000 pueden crear un sesgo contra los cambios, lo mejor es ponerlos en práctica después de haber realizado las mejoras fundamentales.[44] Del mismo modo, cuando los programas de CMM se implementan con un enfoque en la documentación y la conformidad hacia una manera particular de hacer el trabajo, pueden estandarizarse en base a prácticas poco ideales y crear una tendencia en contra del cambio. Por lo tanto, pueden ser mejor implementadas por separado a partir y después de mejorar los procesos.
Una tendencia contra el cambio del proceso no es el problema más difícil a enfrentar con programas tales como ISO9000 y CMM. En su aplicación, con frecuencia estos programas tienden a eliminar el diseño del proceso y la autoridad de toma de decisiones de los desarrolladores y ponerlo bajo el control de los organismos centrales. Los desarrolladores a menudo equiparan éste problema asumiendo el rol de ingenieros industriales de la época de la administración científica que conocían la mejor manera para que puedan hacer su trabajo. El Pensamiento Lean aprovecha la inteligencia de los trabajadores de primera línea, creyendo que ellos son los que deben determinar y mejorar continuamente la forma de efectuar el trabajo.
Watts Humphrey, quien dirigió el desarrollo inicial de CMM, cree que el desarrollo de software no puede tener éxito sin gente disciplinada y motivada.[45] Esta afirmación evidentemente es correcta, pero se debe considerar también la idea de no centrarse en hacer las cosas bien la primera vez, ya que no es un enfoque apropiado para un entorno de diseño. En cambio, la experimentación y la retroalimentación son más eficaces.[46] El factor crítico en la motivación no es la medición, sino el potenciamiento,[47] basado en la delegación de decisiones hasta nivel más bajo posible de una organización mientras se va desarrollando la capacidad en las personas de tomar decisiones con mejor juicio.

5.5.3. CMMI

El modelo CMM fue reemplazado por una mejora denominada Integración de Modelos de Madurez y Capacidades (CMMI) a finales de 2003. Después de desarrollar varios modelos de madurez, el Instituto de Ingeniería de Software (SEI) los combinó dando lugar al CMMI, que promueve una única descripción genérica de madurez para el desarrollo de software, ingeniería de sistemas, desarrollo de productos y otras disciplinas. La naturaleza específica del software de las Áreas Clave del Proceso de CMM dará paso a una medida genérica de madurez.
La definición de madurez CMMI se basa en dos supuestos:

Supuesto 1: Un sistema se maneja mejor mediante su descomposición en productos de trabajo identificables que se transforman a partir de una entrada a un estado de salida para lograr los objetivos específicos.[48]

Supuesto 2: Una organización madura es aquella en la que todo está cuidadosamente planificado y controlado para cumplir con el plan.

Estos argumentos suenan más bien como una gestión científica; pero se podría tomar un modelo diferente de lo que significa la madurez:

Supuesto Lean 1: Una organización madura mira a todo el sistema; no se centra en la optimización de las piezas desagregadas.[49]

Argumento Lean 2: Una organización madura se centra en el aprendizaje de manera efectiva y faculta a las personas que hacen el trabajo para tomar decisiones.

Se esperaría que una organización que respeta a los desarrolladores de software como profesionales diseñe sus propios puestos de trabajo en base a una formación adecuada, entrenamiento y asistencia, permitiendo mejorar continuamente la forma de hacer su trabajo como parte del proceso de aprendizaje y dándoles el tiempo y equipo necesario para cumplir con los objetivos.
En una organización Lean, las personas que agregan valor son el centro de energía de la organización. Los trabajadores de primera línea tienen la autoridad para diseñar los procesos y la responsabilidad de tomar decisiones, siendo el centro de los recursos, información y entrenamiento.

5.5.4. AUTODETERMINACION

5.5.4.1. El MISTERIO DE LA NUEVA UNIDAD DE FABRICACION DE MOTORES (NUMMI)

En 1982, General Motors (GM) cerró su planta de Freemont, California. La productividad fue de las más bajas de cualquier planta de GM, la calidad era pésima. El ausentismo era tan alto que la planta empleaba el 20 por ciento más de trabajadores de lo necesario sólo para asegurar un esfuerzo de trabajo adecuado en un día determinado.
Dos años más tarde, la misma planta fue reabierta por Nueva Unidad de Fabricación de Motores Inc. o NUMMI, una empresa conjunta entre Toyota y GM. Toyota gestionó y administró la planta, pero estaba obligado a volver a contratar a los ex empleados de GM.El ochenta y cinco por ciento de los trabajadores por hora eran de la antigua planta de GM, incluyendo toda la dirigencia sindical.
En dos años, la productividad de NUMMI fue mayor que cualquier planta de GM, doblando la de la planta original. La calidad era mucho más alta y casi igualó a las plantas japonesas de Toyota. El ausentismo se redujo alrededor del 3 por ciento y el 90 por ciento de los empleados se describía a sí mismos como “satisfechos”.[50]
Es evidente que algo en las prácticas de gestión hizo toda la diferencia a los empleados NUMMI, y esas prácticas han sido sostenibles. A medida que la planta supera los 30 años de funcionamiento, aún sigue encabezando todas las demás plantas de GM en la productividad y la calidad, mientras que la satisfacción del empleado sigue siendo muy alta. Otras plantas de GM han sido capaces de copiar las prácticas de gestión de NUMMI, aunque otras plantas gestionadas por Toyota en los Estados Unidos han obtenido un éxito con resultados similares.
De todos modos, el momento del ensamble de los automóviles sigue siendo una tarea difícil y un trabajo repetitivo. En la planta de NUMMI, los trabajadores repiten las mismas acciones

aproximadamente una vez por minuto, y durante ese minuto, ellos están ocupados durante 57 segundos. En el pasado, se trabajaba por sólo 45 segundos de cada minuto, por lo que ahora trabajan mucho más duro y hacen exactamente lo mismo cada vez. Esto puede sonar reglamentado, lo cual es una realidad. El trabajo fue reglamentado también en la antigua planta de GM. De hecho, había 80 ingenieros industriales que concurrían con contadores de tiempo para diseñar cada tarea. Entonces estos les transmitían a los trabajadores exactamente cómo hacer la tarea, lo cual no los dejaba conforme.
La primera tarea que los directivos de la planta NUMMI hicieron fue conseguir cronómetros para todos y se enseñó a los trabajadores cómo diseñar sus propios puestos de trabajo. Todo el trabajo en NUMMI se realiza en equipos de seis a ocho personas, uno de los cuales es el líder del equipo. El equipo diseña sus propios procedimientos de trabajo, coordinando las normas con los equipos que realizan la misma tarea en turnos alternos. El rol de la dirección es encargarse de entrenar y ayudar a los equipos. Los Ingenieros están disponibles si los trabajadores quieren hacer un pedido o una consulta a ellos, pero en el fondo, cada equipo es responsable de sus propios procedimientos y su calidad.
Se considera que el traslado de las *prácticas* de un entorno a otro es a menudo un error. En su determinado puesto de trabajo se tiene que entender los principios fundamentales de las prácticas y transformar los principios en nuevas prácticas para un nuevo entorno. De hecho, Toyota no transfirió las prácticas de producción japonesas *en masa* a NUMMI. Pero se produjo una transferencia de convicciones en donde lo fundamental es el respeto humano proporcionando un ambiente donde los trabajadores puedan participar activamente en la gestión y la mejora de sus áreas de trabajo y sean capaces de utilizar plenamente sus capacidades.

5.5.4.2. UN PROCESO DE MEJORA DE LA GESTION

Las Organizaciones actuales están plagadas de programas de mejora fallidos, que van desde CMM, ISO9000, TQM, Six Sigma o incluso Lean. Implementar programas de mejora exitosos es notoriamente difícil, y más aún sostenerlos en el tiempo. El programa *Work-Out*, desarrollado originalmente en General Electric, es diferente. Fue concebido como una forma de cambiar el comportamiento de los mandos medios y liberar los conocimientos específicos de las personas más cercanas al trabajo.
En una sesión de *Work-Out*, cincuenta o más trabajadores se reúnen durante dos o tres días y presentan propuestas que les ayuden a hacer un mejor trabajo. Los equipos exponen propuestas concretas para eliminar los procesos que se interponen en el camino e implementar prácticas que aporten valor más rápido. Antes de que el *Work-Out*haya finalizado, se requiere que los administradores aprueben o rechacen cada propuesta presentada, bien sea en el acto o dentro de un mes, y se espera que aquellos que hicieron propuestas sean responsables de su implementación. La combinación de simples herramientas, ejecución inmediata y la participación de prácticamente todos los miembros de la empresa se combinan para hacer de *Work-Out*un programa de mejora particularmente exitoso.
En la mayoría de los programas de mejora, los administradores dicen a los trabajadores cómo hacer su trabajo. En *Work-Out* se cambian los roles; los trabajadores dicen a los administradores cómo dejar que ellos realicen su trabajo. *Work-Out* es un proceso que enseña a los gerentes a escuchar a los trabajadores y tomar medidas en base a sus sugerencias, consultando siempre a los niveles superiores de autoridad para asegurarse de

que lo hagan sin demora. *Work-Out* asume que los trabajadores saben hacer su trabajo y centra su atención en cambiar los sistemas que, a la vista de los trabajadores, les impiden hacer un buen trabajo.

5.5.5. MOTIVACION

5.5.5.1. MAGIA EN 3M

De vez en cuando, un grupo de personas se agrupan para lograr algo grande, donde todo el mundo está completamente comprometido en la tarea dedicada a ese fin. La pasión y el compañerismo crean una atmósfera intensa en la que todo es posible.
3M es una de las pocas grandes empresas en las que la magia de los equipos es fácil de encontrar. En cualquier momento dado hay decenas de grupos energizados y auto-organizados que trabajan en la comercialización de nuevos productos. Como resultado de esto, 3M tiene uno de los registros más grandes de introducción de nuevos productos en el mundo, donde el 30 por ciento de las ventas logradas de cada división es generado por los productos introducidos en los últimos 4 años. El torrente de nuevos productos ha mantenido a la empresa ampliamente diversificada y continuamente renovada por décadas. Esto ha estado sucediendo desde hace más de 75 años.
3M tiene una fórmula simple y muy eficaz que permite que el espíritu empresarial florezca. Esta fórmula fue puesta en marcha por William McKnight, quien dirigió la empresa a través de su formación y crecimiento entre los años 1930 y 1950. A pesar de que nunca estuvo en un nuevo equipo de producción, creó el alma de una nueva máquina de desarrollo de productos. En esencia, son pequeños grupos de auto-organización que se convierten en una posibilidad y se les permite que sea una realidad. La visión de McKnight se resume en cuatro frases: [52]

- “Contratar buenas personas y luego dejarlas en paz”.
- “Si se pone vallas alrededor de la gente, se consiguen ovejas. Hay que darle a la gente el espacio que necesitan”.
- “Alentar, no frenar. Dejar a la gente correr con una idea”.
- “Dar oportunidades, y rápido”.

3M pone un gran valor en la investigación científica y anima a todos sus científicos a la búsqueda de nuevas oportunidades de productos. Se espera que los científicos pasen el 15 por ciento de su tiempo en proyectos de su propia elección, preferentemente nuevos proyectos de desarrollo de productos. En los tiempos de holgura se crea un ambiente en el cual las personas tienen el tiempo para jugar con nuevas ideas de productos. La empresa cuenta con una amplia gama de foros para que los científicos puedan conocerse, intercambiar conocimientos y debatir ideas interesantes.
También existen múltiples programas de reconocimiento que premian a los científicos por sus contribuciones en nuevos productos exitosos. Este ambiente anima a los grupos a formarse espontáneamente en torno a una nueva idea de producto.
Los nuevos programas generalmente comienzan con una persona que tiene una idea para un nuevo producto; se reclutan voluntarios y se reúnen los recursos suficientes para probar su idea. El ambiente de trabajo fomenta la reunión de los miembros de los equipos que se

forman alrededor de las ideas convincentes, y las ideas tienen una tendencia a inspirar al equipo. En primer lugar, la tecnología se perfecciona e invariablemente se realizan invenciones. Durante este tiempo de formación, el grupo probablemente adquirirá unos pocos gerentes patrocinadores que previsiblemente seguirán el proyecto para su supervisión. Los gerentes patrocinadores pueden asistir a los miembros del equipo contratados con acceso a los materiales y equipos de laboratorio. Los productos de muestra son fabricados y probados por clientes potenciales.
Cuando el equipo ha realizado el trabajo suficiente para obtener la condición de estar apto para funcionar, éste tiene que superar tres obstáculos simples: El producto debe satisfacer una necesidad real, debe usar la tecnología de 3M y debe tener un buen potencial de ganancias. Dos barreras comunes no están presentes: No existe un umbral de ingresos y no se tiene la necesidad de un ajuste estratégico de las empresas ya existentes. Los miembros originales del equipo continúan avanzando, apuntando a llegar a la comercialización del producto. Si termina creando un negocio exitoso, pueden esperar a terminar de ejecutarlo, o bien, pueden volcar el negocio en una división a ejecutar, para luego volver a crear un nuevo producto.
La invención crítica que permitió que todo esto ocurra fue la idea de McKnight de una organización que evoluciona continuamente de la creatividad y la iniciativa de los empleados y no de la planificación estratégica de los directivos.

5.5.5.2. PROPOSITO

Las personas necesitan más que una lista de tareas. Si su trabajo consiste en proporcionar motivación intrínseca, tienen que comprender y comprometerse con el objetivo de la obra. La motivación intrínseca es especialmente poderosa si las personas de un equipo se comprometen juntas para lograr un objetivo que les interesa. Hay muchas cosas que se puede hacer para ayudar a un equipo a ganar y mantener un sentido de propósito:
Comenzar con un propósito claro y convincente: Los equipos exitosos en 3M tienen siempre un personaje cuyo primer trabajo es comunicar una visión convincente del potencial del nuevo producto con el fin de contratar voluntarios. Los miembros del equipo que se comprometen a un propósito convincente colaborarán con entusiasmo para crear un nuevo producto en el mercado.
Asegurarse de que el objetivo es alcanzable: La regla fundamental de la capacitación es asegurar que el equipo tenga en sí mismo la capacidad de lograr el propósito de su trabajo. Si un equipo se compromete a llevar a cabo un objetivo de negocio, debe tener acceso a los recursos necesarios para lograr ese objetivo.
Dar al equipo acceso a los clientes: Es una manera para que los miembros del equipo puedan entender el propósito de lo que se está proyectando o realizando. Es significativo si se puede ver cómo el software va a hacer la vida más fácil para la gente real. Esto también da a los miembros del equipo una idea de cómo su trabajo individual encaja en el cuadro general.
Dejar que el equipo haga sus propios compromisos: Al comienzo de una iteración, el equipo debe negociar con los clientes para entender sus prioridades y seleccionar los trabajos para la próxima iteración. Nadie debe pretender decir al equipo la cantidad de trabajo que debe ser capaz de terminar. Cuando los miembros se comprometen a una serie de características, están asumiendo un compromiso con el otro.

El rol de la gerencia es realizar interferencia: Un equipo altamente motivado no necesita que le digan lo que debe hacer, pero es posible que tenga que dar a sus líderes un informe de avance. Probablemente necesitarán algunos recursos. Los líderes pueden no ser capaces de satisfacer todas las peticiones, pero el equipo tendrá que mantener el ritmo del trabajo y sus miembros reconocen que alguien los supervisa.
Mantener a los escépticos alejados del equipo: Nada mata un objetivo más rápido que alguien que sabe que no se puede hacer y tiene muchas razones de por qué. El equipo no necesita oírlo.
Se puede imaginar a un equipo de desarrollo de software como un polígono de varios lados (Figura 9.1), donde cada lado del polígono tiene los objetivos a cumplir. Mientras los clientes buscan que el sistema entregue valor de negocio, los analistas o gestores de productos ayudan a los clientes a articular las características en detalle y hacerlas comprensibles para los desarrolladores. Los desarrolladores estiman la cantidad de tiempo necesario para entregar el software en funcionamiento. Los que se encargan de probar o testear el producto ayudan a asegurar que el sistema cumple con las necesidades de los clientes mediante la creación de pruebas de cliente. Apoyar a las personas ayuda a la implementación y capacitación de los usuarios, y asegura que el servicio de asistencia conoce cómo responder a las inquietudes. En conjunto, éste equipo tiene un objetivo: brindar valor de negocio.

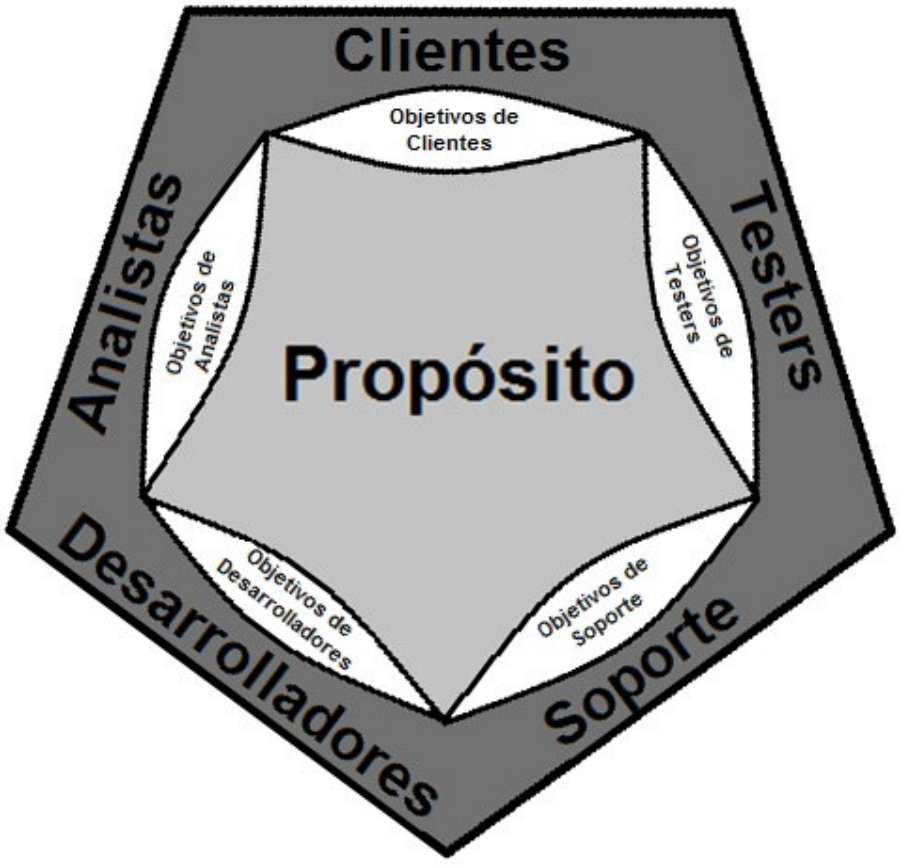


Figura 5.9 El polígono del equipo

El número de lados del polígono y las disciplinas específicas necesarias para lograr un propósito variará dependiendo del tipo de proyecto. Algunos clientes no necesitan analistas, mientras que otros necesitan ayuda para traducir una visión a grandes rasgos en detalles para que los desarrolladores puedan trabajar. A veces los encargados de realizar las pruebas pueden asumir el papel de analista, y viceversa. Los desarrolladores con conocimientos del dominio suelen servir en el rol del analista. Lo que importa es que los analistas no se interpongan en el camino de una comunicación directa entre el desarrollador y el cliente, sino que facilite la comprensión de ambas partes.

5.5.5.3. BLOQUE DE MOTIVACION

La motivación intrínseca se maneja por medio de autodeterminación y un sentido de propósito, pero no prosperará en un clima hostil. Investigaciones han demostrado que la motivación intrínseca requiere un sentimiento de pertenencia y seguridad, además de un sentido de competencia y progreso.[53]

5.5.5.4. PERTENENCIA

En el ambiente de trabajo actual se necesita un equipo para lograr la mayoría de los propósitos. En un equipo saludable, cada integrante tiene claro cuál es el objetivo y se compromete para su éxito. Los miembros del equipo se respetan mutuamente y son honestos con los demás.Por último, el equipo debe ganar o perder como un grupo. Dar crédito a un individuo por esfuerzos que son del equipo y fomentar competencia para crear ganadores y perdedores es una buena manera de matar la motivación del equipo. Si sólo unos pocos miembros logran ser ganadores, los otros integrantes aprenden a preocuparse únicamente por sí mismos y no por el bien general del equipo.

5.5.5.5. SEGURIDAD

Una de las maneras más rápidas para mata la motivación es lo que en el ejército de los Estados Unidos es llamado *mentalidad de cero defectos.* Es un clima que no tolera absolutamente ningún error y se requiere perfección hasta el más mínimo detalle. El ejército considera ésta mentalidad un serio problema de liderazgo porque mata la iniciativa necesaria para el éxito en el campo de batalla.
A medida que un negocio crece, se vuelve cada vez más necesario delegar responsabilidades y alentar a las personas a ejercer su iniciativa; y para lograrlo se requiere una considerable tolerancia. Si las personas a quienes se les delega autoridad y responsabilidad son buenas, querrán hacer su trabajo a su manera. Es posible que se cometan errores, pero si esencialmente la persona estaba en lo correcto, a largo plazo los errores no serán tan graves como los cometidos por la gerencia si se obliga a trabajar bajo su autoridad y dictando exactamente cómo deben hacer su trabajo.

5.5.5.6. COMPETENCIA

Las personas deben creer que son capaces de hacer un buen trabajo, es por eso que desean participar en algo ellos creen que va a funcionar. Es muy motivador formar parte de un equipo de trabajo exitoso, pero es muy desmotivador creer que el fracaso es inevitable. Un ambiente de trabajo sin disciplina no genera una sensación de libertad, por el contrario, crea una sensación de fracaso.
Los entornos de desarrollo de software deben ser disciplinados para que el trabajo pueda transcurrir sin problemas, con rapidez y de forma productiva. Se requieren buenas prácticas básicas como normas de codificación, un proceso de compilación y pruebas automatizadas para concretar un rápido desarrollo.
También es importante un mecanismo para el intercambio de ideas y la mejora de los diseños, tal vez mediante el uso de programación en parejas, revisiones de código o enfoques similares.
Un sentido de competencia nace a partir de conocimientos y habilidades, retroalimentación positiva, estándares altos y el cumplimiento de un reto difícil. No obstante, un líder que delega y confía en los trabajadores debe verificar que están en el camino correcto y proporcionar la orientación necesaria para que puedan alcanzar el éxito.

5.5.5.7. PROGRESOS

Incluso un equipo altamente motivado sólo funcionará mientas sus miembros necesiten sentir que han logrado algo. Esto reafirma el propósito y mantiene a todos con entusiasmo. Si no hay ninguna otra razón para desarrollar software en iteraciones, ésta es una razón de peso por sí misma. En cada iteración, el equipo tiene que poner su mejor esfuerzo frente a los clientes y conocer cómo se ha realizado cada iteración. Por supuesto, existe cierto riesgo de que el cliente no quede conforme, pero es mejor averiguarlo anticipadamente y no tarde. Muy a menudo los clientes se complacen de ver software funcional que pueden utilizar realmente, realzando el sentido del trabajo y revitalizando el equipo.

5.5.6. LIDERAZGO

John Kotter en su libro "*Lo que los líderes realmente hacen"* escribió que "Nadie ha descubierto todavía la manera de dirigir eficazmente a las personas en una batalla; deben ser guiados". Kotter establece una clara distinción entre gerentes y líderes, que se resumen en siguiente cuadro:

Gerentes	Lideres
Enfrentan la complejidad	Enfrentan el cambio
Planear y Presupuestar	Establecen el rumbo
Organización del Personal	Alinear a las personas
Seguimiento y Control	Brindar Motivación

5.5.6.1. LIDERES RESPETADOS

Al desarrollar nuevos productos, la innovación se logra mediante equipos motivados y entusiasmados, y si se mira detrás de este equipo, seguramente se encontrará un líder apasionado que escribió el concepto inicial del producto, reunió apoyo gerencial para el programa y eligió a la mayoría del equipo. El líder interpreta la visión del producto para el equipo, representando al cliente que aún no es consciente del nuevo producto. Marca el ritmo del desarrollo y determina cómo tomar las decisiones. También se espera que un líder continúe trabajando en una buena idea, incluso si el programa es cancelado por la dirección.
Un rol similar al descripto anteriormente es desempeñado por el ingeniero en jefe de Toyota, que pasa tiempo estudiando el mercado al que se apunta, y escribe el documento con el concepto del vehículo, establece el diseño general y el calendario, es responsable en última instancia de la performance económica del vehículo. A diferencia del rol de coordinación de un director de un nuevo vehículo en las compañías automotrices de Estados Unidos, el ingeniero en jefe tiene completa responsabilidad del vehículo y la autoridad necesaria para tomar todas las decisiones del programa. El ingeniero en jefe de Toyota ha sido llamado como un *director del programa de alto nivel*,[54] pero éste es un término erróneo, ya que un ingeniero en jefe es mucho más un líder que un director. Tal vez la correcta caracterización de un ingeniero en jefe es un *líder respetado*. El énfasis de su papel se centra en el establecimiento de la dirección, la alineación de la organización y la motivación del equipo.
El ingeniero en jefe de Toyota tiene un fuerte sentido de propiedad del producto, al punto tal que a menudo éste lleva su nombre. Podría parecer que un fuerte sentido de propiedad llevaría al ingeniero en jefe a ejercer un gran control sobre el desarrollo de su producto, pero no tiene autoridad directa sobre las personas que trabajan en el mismo. Es plenamente consciente de que el aprovechamiento de los talentos de un gran grupo de expertos es mucho más efectivo que tratar de controlar el trabajo. Por lo tanto, conduce al equipo de desarrollo en lugar de tratar de manejarlo.

5.5.6.2. DESARROLLADORES MAESTROS

En un amplio estudio de diseño de grandes sistemas,[55] Bill Curtis encontró que para la mayoría de estos, un diseñador o grupo pequeño de diseñadores excepcionales emergen para asumir la principal responsabilidad del diseño. Ejercen el liderazgo a través de su conocimiento superior en lugar de la autoridad otorgada. Su conocimiento profundo tanto de los clientes como de las cuestiones técnicas les otorga el respeto del equipo de desarrollo. Los diseñadores excepcionales son personas que están muy familiarizados con el dominio de la aplicación y son expertos en comunicar su visión técnica al equipo de desarrollo.
Se puede observar que el diseñador excepcional identificado por Curtis tiene las mismas características que un *líder respetado* o un ingeniero en jefe. En el desarrollo de software se han utilizado términos como ingeniero de sistemas, jefe de programadores y arquitecto para designar el rol de un *diseñador excepcional*. Para el tema tratado, se utilizará el término desarrollador maestro para designar el papel de *líder respetado* de un proyecto de desarrollo de software.
El papel de *desarrollador maestro* es esencial; no es necesario identificar un desarrollador maestro al inicio de cada proyecto. Para sistemas pequeños, el desarrollador maestro tiene tendencia a surgir en un equipo auto-organizado. Incluso en los grandes sistemas, Curtis

encontró que los diseñadores excepcionales ejercían el liderazgo en base a su conocimiento y no porque fueran líderes designados. Si se designa un desarrollador maestro, hay que asegurarse de que la persona es un líder respetado que potenciará al equipo. Un jefe que no colabore con los desarrolladores puede impedir el surgimiento del tipo adecuado de liderazgo en el diseño.
Los desarrolladores maestros cuentan con una amplia experiencia en el dominio y la tecnología, comprendiendo tanto a los clientes como a los desarrolladores. Ellos entienden las limitaciones del sistema, las interacciones, los requisitos no declarados, las condiciones de excepción, y la dirección probable del cambio. Ven al sistema con un alto nivel de abstracción, pero sin embargo, pueden profundizar en la complejidad y detalle que los desarrolladores y los clientes deben hacer frente. Tienen la sabiduría para detectar las ventajas del mercado en el desarrollo de productos y las ventajas comerciales en el desarrollo interno. Si un equipo de desarrollo no cuenta con este tipo de líder, buscara uno, ya que los equipos entienden que este tipo de liderazgo es la clave para lograr que sus esfuerzos sean exitosos.
Dado que los desarrolladores maestros son percibidos como personas con mayor conocimiento, se convierten en el punto focal de la comunicación.[56] Las organizaciones con arquitectos que prestan servicio en una función de asesoramiento encontrarán que éstos no encajan en el rol que se define. Los desarrolladores maestros son parte del equipo y están sumergidos en los detalles del trabajo. Proporcionan el liderazgo necesario para tomar buenas decisiones, avanzar rápidamente y desarrollar software de alta calidad.

5.5.6.3. ¿COMO SURGEN LOS DESARROLLADORES MAESTROS?

Los desarrolladores maestros asumen su rol mediante una amplia experiencia en la tecnología y en el dominio al que apunta el sistema, junto con excelentes habilidades de abstracción y comunicación. La experiencia no puede ser sustituida, ya que desarrollar software hábilmente es como aprender un oficio. Los nuevos programadores se inician igual que aprendices de maestros artesanos. A medida que van calificando, enseñan a otros aprendices y finalmente logran trabajar conjuntamente con otro maestro artesano. Así comienza a difundir ideas y desarrollar habilidades generales, llegando por sí mismo a ser un maestro artesano.
Los líderes sólo prosperan en las organizaciones que desean que estén allí. Una organización tiene que valorar el liderazgo para poder desarrollar líderes. Deben tener un programa integrado de aprendizaje de liderazgo, en los que la gente puede aprender el *arte* de la dirección.
Las personas responden a las expectativas de su gestión. Los líderes de desarrollo de software no prosperarán en una organización que valora el proceso, la documentación y la conformidad con el plan por encima de todo. Una organización obtendrá lo que valora y el *Manifiesto Ágil*[57] aporta un gran servicio en el cambio de la percepción de valor de proceso a personas, de documentación a código, de contratos a colaboración y de planes a acción.

5.5.6.4. GESTION DE PROYECTOS

Si analizamos el papel de un director de proyecto en el desarrollo ágil, a menudo, éste no tiene un perfil técnico y en general, no es responsable de desarrollar un profundo

conocimiento de los aspectos técnicos del proyecto. Por lo tanto, el director del proyecto no suele desempeñar el rol de desarrollador maestro.
Por otro lado, el desarrollo ágil se basa en iteraciones cortas en las que los miembros del equipo hacen sus propios compromisos y supervisan su propio progreso en función del cumplimiento de esos compromisos. Aunque una lista de características de alto nivel puede estar organizada en un plan de iteraciones de largo alcance, este plan no conduce el trabajo diario. Los sistemas pull estructuran el trabajo por sí mismo para indicar a los desarrolladores lo que se debe hacer, por lo que en un entorno de desarrollo *Lean* adecuadamente estructurado, un director de proyectos no asigna tareas o supervisa su realización.
En función a lo expuesto, si el equipo está facultado para tomar sus propias decisiones, ¿cuál es el trabajo del director del proyecto?
Las herramientas desarrolladas a lo largo de este trabajo ayudan a definir el papel del liderazgo de proyectos en el desarrollo de software ágil. Los líderes de proyectos comienzan identificando residuos y bosquejan un mapa de flujo de valor del proceso de desarrollo, y abordan los mayores cuellos de botella. Coordinan las reuniones de planificación de iteraciones y las reuniones diarias de estado de avance, proporcionan radiadores de información y ayudan al equipo a obtener los recursos necesarios para cumplir con los compromisos. Coordinan varios equipos asegurando que la sincronización entre ellos sea regular y completa. Aseguran que el entorno de desarrollo cuente con las herramientas estándares, tales como el control de código fuente y pruebas automatizadas, y comprueban que las pruebas de aceptación integrada y refactorización se están llevando a cabo. Trabajan con el área contable para crear modelos financieros para que el equipo pueda tomar buenas decisiones de equilibrio y proveen un ambiente motivador.
Los líderes de proyectos juegan un papel importante en un proyecto ágil. En lugar de planificar utilizando diagramas de Pert y Gantt, crean un plan de lanzamiento con hitos frecuentes y mantienen el foco en la satisfacción de los compromisos de cada iteración. En lugar de preocuparse por el cambio del alcance, se preocupan por la elegancia del cambio, en lugar de preocuparse por los procesos de aprobación de cambio, se preocupan por las prácticas de diseño tolerantes al cambio. Se aseguran de que las pruebas y la integración sean parte del desarrollo en lugar de un evento separado y tardío. Se cercioran de que las personas involucradas en la implementación, capacitación y soporte al cliente estén plenamente involucradas desde el principio.

Capacitación en Gestión de Proyectos Lean

La mayor parte de los temas tratados en un curso tradicional de gestión de proyectos no son lo que un líder de proyecto ágil necesita saber. Se recomienda un conjunto de herramientas alternativas para los líderes del proyecto:

Detectar Residuos	Costos de los retrasos
Ver el Mapa del Flujo de Valor	Autodeterminación
Retroalimentación	Motivación
Iteraciones	Liderazgo
Sincronización	Especialización
Desarrollar Basado en Conjunto	Integridad Percibida
Opciones de Pensamiento	Integridad Conceptual
Último Momento Responsable	Refactorización
Toma de Decisiones	Pruebas
Sistemas Pull	Mediciones
Teoría de Colas	Contratos

5.5.7. ESPECIALIZACION

5.5.7.1. COMUNIDADES DE ESPECIALIZACION

El desarrollo de software es una tarea compleja, con muchas áreas de conocimiento especializado. Por un lado, está el conocimiento técnico (expertos de bases de datos, expertos en la interfaz del usuario, expertos en código integrado y expertos en middleware). Por otro lado, hay una gran cantidad de conocimiento del dominio, como por ejemplo en el desarrollo de un software de atención médica o de seguridad, en cuyos casos es importante desarrollar experiencia. Es necesario tener áreas de especialización de la organización. Si se pretende tener ventaja competitiva en el mercado, es necesario tener en la organización áreas de especialización inexistentes en la competencia. Incluso si la organización brinda servicios a un cliente interno dentro de la misma empresa, sería conveniente entender que experiencia particular que no se puede conseguir por contratación externa aporta el grupo.
La manera tradicional de desarrollar comunidades de especialización en una empresa es dividir la organización en funciones que responden a las competencias básicas necesarias. Cada función contrata y capacita a las personas, establece estándares y desarrolla conocimientos para su competencia en particular. Las funciones asignan personal a los equipos que desarrollan un producto o un sistema bajo la guía de un líder del programa.
Existen problemas inherentes a esta estructura matricial. En primer lugar, existe la posibilidad de que los trabajadores sientan lealtades divididas cuando tienen dos gerentes para satisfacer, y en segundo lugar, existe el peligro de que un lado de la matriz domine al otro. Sin embargo, existen estructuras matriciales exitosas en muchas empresas, y observando en detalle a éstas, se revela que el éxito está determinado por la forma en que los gerentes ven sus trabajos. En empresas con gestión matricial exitosa, los gerentes funcionales ven sus puestos de trabajo como mentores y maestros. Se aseguran de que hay maestros que ayudan a desarrollar trabajadores y aprendices mediante una serie progresiva de asignaciones de trabajo con el apoyo y el entrenamiento adecuado. Al mismo tiempo, los

líderes de los equipos ven sus trabajos como facilitadores y motivadores que reúnen los recursos y eliminan los obstáculos, y como guías que representan la voz del cliente al equipo.
La organización de desarrollo de productos de Toyota es una organización matricial. Los ingenieros en jefe lideran el equipo, pero la profunda experiencia técnica necesaria para diseñar un automóvil reside en las funciones. Los ingenieros permanecen en la misma función quizás por una década antes de ser considerados ingenieros de motor o de diseño realmente experimentado. Durante ese tiempo son asesorados e instruidos por los gerentes que son expertos en su área. Los gerentes funcionales en Toyota son respetados en sus campos y tienen la autoridad para actuar como contrapeso de un ingeniero en jefe en momentos en que se deben tomar importantes decisiones de equilibrio.
Las estructuras de organización matricial son muy útiles para proporcionar comunidades de especialización, pero incluso si una empresa no utiliza una estructura de matriz, es imprescindible contar con comunidades de especialización. El primer paso es identificar las competencias técnicas específicas del dominio que son críticas para el éxito de la organización. Estas pueden incluir capacidades tales como la administración de bases de datos, diseño de interfaz de usuario, seguridad, arquitectura, programación, pruebas y análisis de seguridad. Numerosas empresas entonces crean foros para estas comunidades, y si no hay suficientes personas en un área crítica para formar una comunidad interna, entonces las comunidades externas de conocimientos suelen estar disponibles para cubrir tales áreas.

5.5.7.2. ESTANDARES

El desarrollo de software necesita estándares. Estándares de nomenclatura, normas lingüísticas, estándares de construcción y así sucesivamente son en mayor o menor medida necesarios para que un equipo de desarrollo funcione bien. Las normas suelen ser desarrolladas por la comunidad de especialización correspondiente o, cuando es necesario, por el equipo de programación. Sin embargo, por lo general es mejor para un equipo de programación trabajar con estándares existentes en vez de desarrollarlos por su propia cuenta. Una manera de descubrir dónde se necesita una comunidad de expertos es identificar donde se carece de normas.
Un cuadro desplegable en 50 estados es un ejemplo de un diseño común de interfaz de usuario, donde los estándares parecen faltar.[58] Tal diseño no sobreviviría si es enfrentado a los usuarios de una compañía destacada por su excelencia, donde el aprendizaje y la experiencia se desarrollan a través de la experimentación y el intercambio de conocimientos. Si se encuentran áreas donde los estándares parecen faltar y el trabajo descuidado es evidente, es conveniente adoptar comunidades especialización para desarrollar estándares. Los desarrolladores valoran los estándares razonables, especialmente si apuntan a desarrollarlos y mantenerlos actualizados. Además los clientes aprecian los estándares aún más.

5.6. SEXTO PRINCIPIO - CREAR INTEGRIDAD

A finales de la década de 1980, Kim Clark, de la Escuela de Negocios de Harvard, se dispuso a examinar la forma en que algunas empresas podrían desarrollar constantemente productos de calidad superior. Centró sus estudios en el mercado automotriz, ya que los automóviles son muy complejos y su desarrollo requiere cientos de personas durante muchos meses. Buscó diferencias críticas entre compañías de rendimiento promedio y alto rendimiento y encontró que la diferencia clave fue algo que llamó ***integridad del producto***. Determinó que la integridad del producto tiene dos dimensiones: la integridad externa y la integridad interna. Estos dos conceptos adaptados al desarrollo de software se llamaron: **integridad percibida** e **integridad conceptual**.[59] El primero significa que la totalidad del producto alcanza un balance que satisface al cliente desde el punto de vista de su funcionamiento, utilidad, confiabilidad y economía. La integridad percibida es afectada por toda la experiencia del cliente de un sistema: como se anuncia, se entrega, se instala y se accede, lo intuitiva que es su utilización, como maneja los cambios de dominio, su velocidad de respuesta, su costo y que tan bien resuelve el problema.
Integridad conceptual significa que los conceptos centrales del sistema trabajan juntos como un todo uniforme y coherente. Los componentes coinciden y trabajan bien juntos, la arquitectura logra un equilibrio eficaz entre la flexibilidad, facilidad de mantenimiento, eficiencia y capacidad de respuesta. Emerge a medida que el sistema evoluciona y madura.
La integridad conceptual es necesaria para la integridad percibida, pero no es suficiente. Si la arquitectura más elegante del mundo no hace un trabajo excepcional satisfaciendo las necesidades del cliente, los usuarios no notarán la integridad conceptual subyacente. Es por esta razón que la arquitectura de un sistema debe evolucionar y madurar. La integridad percibida cambiará con el tiempo, y por lo tanto la arquitectura subyacente también debe hacerlo. A medida que se añaden nuevas características a un sistema para mantener la integridad percibida, la capacidad subyacente de la arquitectura para soportar las funciones de una manera cohesiva también debe ser añadida.

5.6.1. LA CLAVE DE LA INTEGRIDAD

La integridad se logra a través de excelente y detallado flujo de información. La integridad percibida es un reflejo de la integridad del flujo de información entre los clientes o usuarios y los desarrolladores. La integridad conceptual es un reflejo de la integridad del flujo de información técnica.

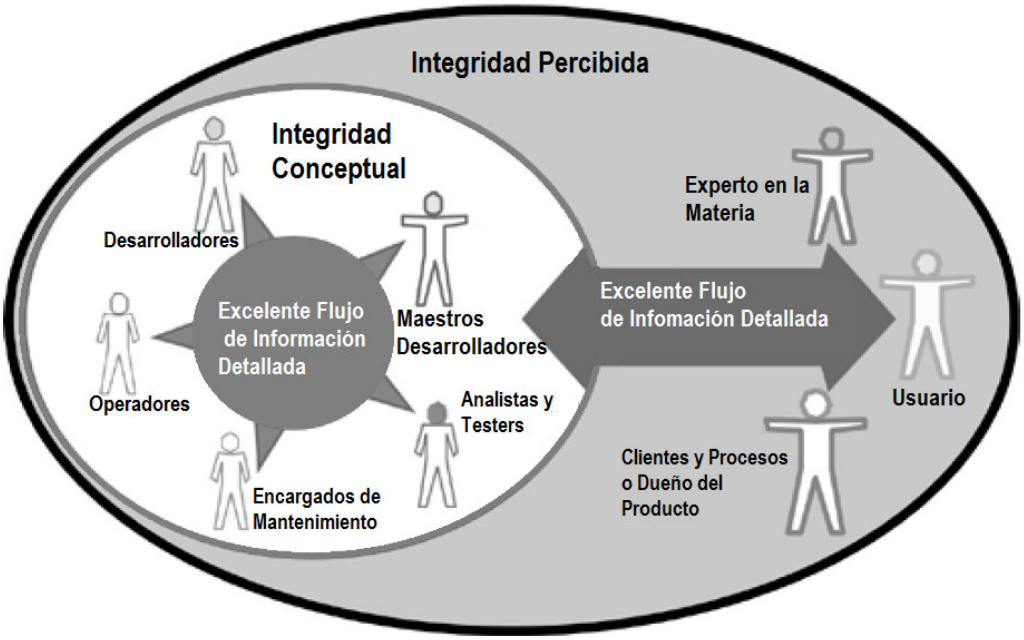


Figura 5.10 El flujo de información produce integridad

La manera de construir un sistema con alta integridad percibida y conceptual es tener un excelente flujo de información entre el cliente y el equipo de desarrollo y entre los distintos procesos con diferentes prioridades para el equipo de desarrollo. El flujo de información debe tener en cuenta los usos actuales y potenciales del sistema.
Esto es consistente con las conclusiones de Bill Curtis y sus colegas que estables que los tres requisitos fundamentales para un mejor desempeño de desarrollo de software son:

Mayor conocimiento del dominio de la aplicación en todo el personal involucrado en el desarrollo.
Aceptación de cambio como un proceso ordinario y la capacidad para dar cabida a las decisiones de diseño emergente.
Un entorno que mejora la comunicación para integrar a las personas, herramientas e información.

5.6.2. INTEGRIDAD PERCIBIDA

Las decisiones que afectan la integridad percibida se hacen todos los días, sobre todo en los niveles más bajos de la organización de desarrollo. Las empresas que logran consistentemente integridad percibida tienen una forma de mantener constantes los valores de los clientes frente a los técnicos que toman decisiones de diseño detallado. En la mayoría de los fabricantes de automóviles japoneses esto se hace por un ingeniero en jefe que ha desarrollado una visión de lo que el segmento de clientes apuntado quiere en un automóvil. El ingeniero en jefe pasa mucho tiempo recorriendo, hablando con los ingenieros y asegurándose de que tienen una buena idea de lo que el cliente encontrará importante. Si la visión de la integridad percibida no se actualiza con regularidad, los ingenieros tienen una tendencia a perderse en los detalles técnicos y olvidar las preferencias de los clientes.

El desarrollo de software secuencial intenta transmitir el concepto de integridad percibida a los programadores a través de un proceso de varias etapas. En primer lugar, los requisitos se obtienen de los clientes y por escrito. Entonces estos requisitos se someten a análisis, generalmente por personas que no son las que los reunieron. El análisis es un intento por comprender en términos más técnicos lo que significan los requisitos y se utiliza para diseñar la manera como se implementará el software. El diseño luego es entregado a otro grupo, los programadores, que se supone que escribirán el código.
Evidentemente este criterio presenta problemas. En primer lugar, los clientes de un sistema de software generalmente no son capaces de definir lo que ellos perciben como integridad. Los clientes saben cuáles son sus problemas, pero muy a menudo no pueden describir la solución. Ellos sabrán si es un buen sistema cuando lo vean, pero no pueden verlo de antemano. Para empeorar esto, a medida que cambien las circunstancias, también lo hará la percepción de los clientes sobre la integridad del sistema.
Los problemas con el desarrollo secuencial no desaparecen incluso si los clientes pueden visualizar y alguien puede documentar un conjunto preciso de requisitos. Tradicionalmente los requisitos son escritos y entregados a un equipo de analistas que realiza un análisis y entrega los resultados a los diseñadores, que diseñan el software y entregan los resultados a los programadores. Son ellos los que día a día tomaran decisiones sobre cómo escribir el código exactamente. Como puede verse, hay dos o tres documentos intermedios hasta lograr la comprensión de los clientes de la integridad del sistema. En cada entrega de estos documentos, una considerable cantidad de información se pierde o mal interpreta, y por lo tanto, detalles claves y futuras perspectivas no se obtienen en primera instancia.
Si un proceso no proporciona un flujo de información precisa y detallada desde el cliente a los desarrolladores, el producto resultante carecerá de integridad percibida. Es difícil imaginar que este tipo de información pueda ser transmitida a través de múltiples iteraciones de documentos entregados a múltiples grupos de personas.
Como alternativa, existen varias técnicas que se pueden utilizar para establecer flujos de información cliente-desarrollador de primer nivel:

Los sistemas más pequeños deben ser desarrollados por un único equipo con acceso inmediato a las personas que van a juzgar la integridad del sistema. El equipo debe utilizar iteraciones cortas y mostrar cada una a una amplia gama de personas que reconocerán la integridad al verla, de esta manera se puede corregir el rumbo basados en la retroalimentación.
Pruebas realizadas por los clientes, lo que ofrece una excelente comunicación cliente-desarrollador.
Un sistema complejo debe ser representado mediante un lenguaje y un conjunto de modelos que los clientes entiendan y los programadores puedan usar sin necesidad de refinamiento.
Los grandes sistemas deben tener un desarrollador principal que tenga profundo conocimiento del cliente y excelentes credenciales técnicas, cuya función es facilitar el diseño, representando los intereses del cliente ante los demás desarrolladores.

Estos enfoques no son mutuamente excluyentes. Incluso los desarrolladores principales de primera categoría se benefician de iteraciones frecuentes. No importa la técnica de comunicación que sea usada, las pruebas de los clientes deben estar preparadas para presentar ejemplos de cómo funciona el sistema. Estas pruebas ayudan a los clientes a

entender cómo se comportará el sistema para que los desarrolladores puedan estar seguros de que su trabajo cumple con las expectativas de los clientes.

5.6.3. DISEÑO BASADO EN MODELOS

Existen muchas maneras de modelar cualquier cosa. La modelización asegura que los resultados serán una representación correcta de las cuestiones de dominio, y que al mismo tiempo serán posibles de implementar al software de manera eficaz. El diseño basado en modelos es un enfoque valioso para los sistemas complejos, ya que permite que todos hablen el mismo idioma.
Los modelos captan cómo el sistema se muestra al usuario, cómo va a lidiar con conceptos y reglas significativas y cómo se van a proporcionar resultados. El tipo de modelo dependerá del ámbito y del modo en que sus datos puedan representarse mejor en un formato conciso.
Se utilizan varios modelos para apoyar un excelente flujo de información cliente-desarrollador en el desarrollo de sistemas complejos:

Un modelo de dominio conceptual: esto podría ser un modelo de clases de las entidades básicas en el sistema, ya sean eventos, documentos, transacciones, representaciones de objetos físicos, etc.
Un glosario: esto define los términos que se encuentran en el modelo de dominio y asegura un lenguaje consistente para el equipo.
Un modelo de casos de uso: el modelo de dominio y el glosario son vistas estáticas del dominio. Un modelo de casos de uso es una visión dinámica del sistema y es útil para capturar conocimiento tácito sobre lo que significa realmente la usabilidad en éste ámbito.
Calificadores: las primeras implementaciones de un sistema a menudo se codifican y se prueban en un entorno de desarrollo, donde es difícil simular todas las interacciones y las cargas que el sistema en funcionamiento podría experimentar. Los desarrolladores deben entender qué multiplicador o cuantificador podría aplicarse a la funcionalidad básica para conseguir valor de negocio.

Los desarrolladores que escriben la capa de negocio y la capa de presentación del código deben utilizar estos modelos directamente, sin necesidad de traducción. Cuando se está hablando del mismo concepto, los clientes y los desarrolladores deben utilizar las mismas palabras, por lo general palabras tomadas del dominio o una metáfora del dominio. Si los modelos se traducen o se utilizan diferentes lenguajes, una gran cantidad de información se pierde o es ilegible. Además, el software que refleje directamente el modelo de dominio será más resistente a las cambiantes necesidades que uno con estructuras internas significativamente diferentes, elegidas por razones puramente técnicas.
Una manera de determinar si un modelo es útil es observar si se mantiene al día. Algunos creen que es importante tener un modelo actualizado para que pueda ser utilizado, pero es todo lo contrario. Cuando un modelo deja de ser útil, no será mantenido. Está bien crear modelos que son útiles durante un tiempo y, finalmente, caen en desuso. Pero es un desperdicio crear y mantener modelos simplemente porque parece una buena idea.
Cuando se utilizan modelos, deben ser vistos con un nivel de detalle apropiado para abordar al cliente o su representante. La mejor manera de hacer esto es empezar con un alto nivel de abstracción y añadir detalles a la hora de iniciar la implementación de un área en particular.

La gente puede hacer frente a un número limitado de conceptos a la vez, por lo que en un sistema de software complejo, la comunicación será necesariamente limitada a solo un puñado de conceptos a la vez. La clave de la comunicación acerca de los sistemas complejos es ocultar los detalles detrás de las abstracciones cuando se desea un panorama general y pasar a niveles más bajos de abstracción para profundizar en los detalles. Los modelos son herramientas útiles para la creación de abstracciones y permitir la comunicación en temas más amplios. Las iteraciones son el mecanismo clave para activar el movimiento de abstracciones a la implementación de los detalles.
Las pruebas son la mejor manera de recordar los detalles de lo acordado y asegurar que las características siguen funcionando a medida que el sistema evoluciona y al final de una iteración, el cliente comprueba si el concepto general es aceptable.

5.6.3.1. MANTENIMIENTO DE LA INTEGRIDAD PERCIBIDA

Incluso un buen flujo de información cliente-desarrollador puede no capturar la necesidad estratégica de las aplicaciones para cambiar en el futuro. La mayoría de los sistemas de software son dinámicos en el tiempo y más de la mitad del gasto en una aplicación se producirá después de que entre en producción.[60] Los sistemas deben ser capaces de adaptarse a los negocios y al cambio tecnológico de una manera económica.
Uno de los enfoques clave para incorporar el cambio dentro de una infraestructura de información es asegurarse de que el proceso de desarrollo incorpore por sí mismo los cambios en curso. Uno de los temores que surgen al considerar un enfoque de desarrollo iterativo es que las iteraciones posteriores introducirán capacidades que requieren cambio en el diseño. Sin embargo, si un sistema es construido bajo el paradigma de que todo debe ser conocido por adelantado de manera que el diseño óptimo se pueda encontrar, entonces es probable que no sea adaptable a los cambios en el futuro. Un proceso de diseño tolerante al cambio probablemente dará como resultado un sistema tolerante al cambio.
El mantenimiento de la memoria intuitiva de un sistema es clave para asegurar su integridad a largo plazo. Ha habido muchos intentos de utilizar documentación generada durante el diseño para hacer esto. Sin embargo, la documentación de diseño rara vez refleja al sistema como realmente fue construida, por lo que es ampliamente ignorada por los programadores de mantenimiento. Si este es el único propósito para el que sirve ésta documentación, entonces es un desperdicio crearlo. Una forma de mantener la memoria intuitiva de un sistema es hacer a los desarrolladores responsables de las actualizaciones permanentes. Alternativamente, los desarrolladores y los programadores de mantenimiento pueden trabajar de forma conjunta durante un periodo de tiempo para transferir el conocimiento tácito. También se puede crear un modelo de cómo fue construido el sistema después de ser desarrollado. Pero la mejor manera de mantener el conocimiento intuitivo de un sistema y conservarlo es ofrecer un conjunto de pruebas automatizadas junto con el código, complementado por un modelo general de alto nivel creado al final del esfuerzo de desarrollo inicial.

5.6.4. INTEGRIDAD CONCEPTUAL

Integridad conceptual significa que los conceptos centrales de un sistema trabajan conjuntamente como un todo uniforme y coherente. Los componentes coinciden y trabajan bien juntos, la arquitectura logra un equilibrio eficaz entre flexibilidad, facilidad de

mantenimiento, eficiencia y capacidad de respuesta. La arquitectura de un sistema de software se refiere a la forma en que el sistema está estructurado para proporcionar las características y capacidades deseadas. Una arquitectura eficaz es lo que le da integridad conceptual a un sistema.
Si tomamos nuevamente el ejemplo de la fabricación de automóviles, para lograr la integridad conceptual cientos de ingenieros están involucrados en un periodo de unos tres años. Cientos de partes especializadas son desarrolladas por grupos especializados de ingeniería, se toman miles de decisiones detalladas y se realizan igual cantidad de correcciones a las mismas. La clave para lograr la integridad conceptual del producto en un automóvil es la eficacia de los mecanismos de comunicación desarrollados entre estos grupos en el marco de la toma de decisiones.[61]
Existen dos prácticas clave que utilizan las empresas automotrices para lograr integridad conceptual. En primer lugar, el uso de piezas existentes elimina muchos grados de libertad y por lo tanto reduce la complejidad y la necesidad de comunicación. La segunda práctica utilizada es el uso de resolución integrada de problemas para garantizar un excelente flujo de información técnica. Esto significa que:[62]

La comprensión y la solución del problema ocurren al mismo tiempo, no secuencialmente.
La información preliminar se publicará al principio, el flujo de información no será frenado hasta que la información completa esté disponible.
La información se transmite frecuentemente en lotes pequeños, no de una sola vez en un gran lote.
La información fluye en dos direcciones, no en una solamente.
Los medios preferidos para la transmisión de información es la comunicación cara a cara en lugar de documentos y correo electrónico.

Sin la resolución integrada de problemas, los diseñadores deciden de manera aislada la combinación de características y capacidades que ofrecerán el mayor valor para los clientes. Cuando el diseño se completa, un gran lote de información se envía desde los diseñadores a las personas que deben decidir la mejor manera de desarrollar todo a un costo y velocidad aceptable. Este enfoque puede ser diagramado como en la Figura 10.2

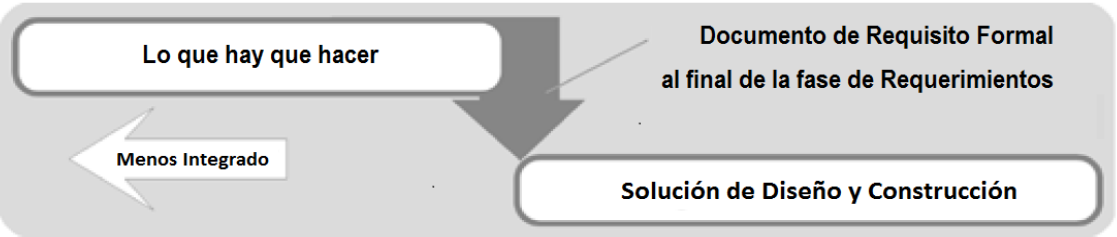


Figura 5.11 Requerimientos antes del diseño

Con la resolución integrada de problemas, ilustrada en la Figura 10.3, el panorama cambia hacia otro con comunicación temprana, frecuente y bilateral. Esta rica comunicación quita énfasis a los mecanismos de control, favoreciendo los debates cara a cara, lotes pequeños, velocidad y flujo.

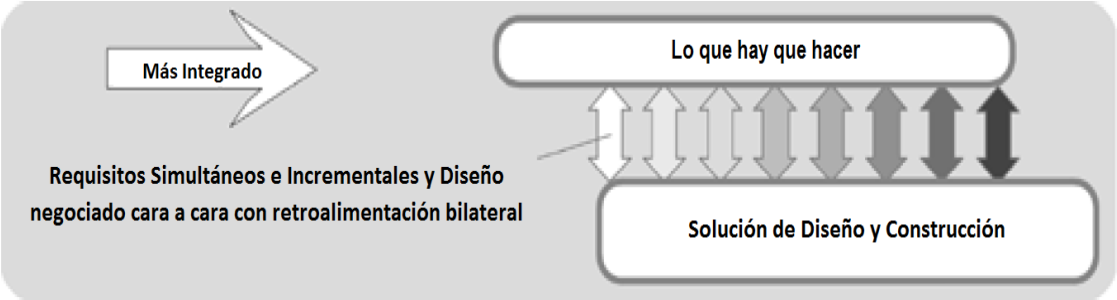


Figura 5.12 Requisitos Simultáneos e Incrementales

5.6.4.1. ARQUITECTURA BASICA DE SOFTWARE

La arquitectura de un automóvil comienza con los conceptos básicos: un motor, un chasis, transmisión, y así sucesivamente. Del mismo modo, la arquitectura de software para la mayoría de los sistemas complejos por lo general comienza con el patrón estándar de capas. Las capas dan una base sólida para la arquitectura del sistema. Los elementos básicos en el desarrollo de software son los siguientes:[63]

- Presentación (interfaz de usuario)
- Dominio (lógica de negocio)
- Fuente de datos (persistencia, mensajería)

Algunos autores identifican capas adicionales:[64]

- Presentación (interfaz de usuario)
- Servicios (gestión de transacciones)
- Dominio (lógica de negocio)
- Traducción (mapeo)
- Fuente de datos (persistencia, mensajería)

Las capas inferiores no deben depender de las capas superiores, por ejemplo, la base de datos no sabe nada de la lógica del negocio y la lógica de negocio es independiente de la interfaz de usuario. Debe ser posible poner a prueba cada capa de forma independiente de las otras mediante la simulación del comportamiento de las otras capas. De ésta manera se proporciona alta cohesión dentro de la capa y separación de funciones e incumbencias entre capas, que son dos patrones arquitectónicos fundamentales en el diseño de software. Estos dos patrones se utilizan iterativamente para lograr la integridad del sistema.
La estructura conceptual de cada capa es otra área que requiere un examen temprano. Se debe prestar particular atención a la capa de presentación, ya que la integridad conceptual en el diseño de la interfaz de usuario es el principal motor de la integridad percibida. Además, puede ser difícil modificar una interfaz una vez que se envió, ya que es más difícil cambiar los hábitos del usuario que cambiar el código.
Una de las funciones de la arquitectura de software es permitir que los sistemas se adapten al cambio de las empresas y al cambio técnico de una manera económica. Dado que no se puede construir un sistema con flexibilidad completa, se debe tratar de agrupar las cosas

que pueden cambiar y ocultarlas del resto del sistema. Esto permite que se hagan cambios que tienen solo un impacto local sin alterar grandes partes del sistema.
Incluso mientras se diseña para dar cabida al cambio, se debe tener cuidado con la tentación de pasar demasiado tiempo analizando lo que el sistema pueda llegar a necesitar en el futuro con el fin de diseñar una gran arquitectura desde el principio. Como hemos visto, la predicción del futuro tiende a ser una pérdida de tiempo y recursos. Es mejor tomar un enfoque en amplitud y obtener los conceptos básicos correctos. Luego se debe dejar emerger los detalles y planear la refactorización regular para mantener la arquitectura saludable.

5.6.4.2. INTEGRIDAD EMERGENTE

¿Cómo se puede estar seguro de que una buena arquitectura surgirá y que el sistema tendrá integridad conceptual? Las prácticas utilizadas en el desarrollo de productos automotrices pueden ser igualmente eficaces en el desarrollo de software.
En primer lugar, se debe utilizar piezas existentes cuando sea posible. Esto significa utilizar software “enlatado” cada vez que se pueda. Al fijar tantos puntos del sistema como sea posible con el software existente y las normas, se reduce la comunicación requerida y se despeja el camino para una mejor comunicación con el resto del sistema.
En segundo lugar se debe utilizar la resolución integrada de problemas. Esto significa iniciar la escritura del software antes de finalizar los detalles del diseño. Se debe mostrar software parcialmente completo a los clientes y usuarios para obtener sus comentarios. Hay que asegurarse que los desarrolladores tienen acceso a los clientes o sus representantes para conseguir preguntas respondidas tan pronto como se presenten. Es necesario ejecutar pruebas de usabilidad en cada característica a medida que se desarrollan.
En tercer lugar, hay que asegurarse de que haya desarrolladores experimentados involucrados en todas las áreas críticas. Ciertamente, no todas las personas que trabajan en el desarrollo de un sistema pueden o deberían tener gran experiencia, pero un sistema complejo requiere que los desarrolladores en el equipo entiendan las complejidades de las diversas áreas técnicas en el sistema, así como los patrones generalmente utilizados para hacer frente a las complejidades.
Finalmente, los sistemas complejos requieren del liderazgo de un desarrollador maestro con las habilidades necesarias para facilitar los esfuerzos de colaboración a través de múltiples equipos de desarrollo. Es importante integrar los esfuerzos para resolver problemas entre sí de acuerdo a las necesidades del cliente. La comunicación necesaria para asegurar que esto ocurra seria responsabilidad del desarrollador maestro.

5.6.5. REFACTORIZACION

.
El historiador de ingeniería Henry Petroski ha escrito extensamente sobre cómo se lleva a cabo realmente el proceso de diseño.[65] Los ingenieros comienzan con algo que funciona, aprenden de sus debilidades y mejoran el diseño. Las mejoras no vienen solo de satisfacer las demandas de los clientes o la adición de características, también son necesarias porque los sistemas complejos tienen efectos que no se entienden bien durante el periodo de diseño. No es razonable esperar un diseño impecable que se anticipe a todas las contingencias posibles o los efectos en cascada producidos por simples cambios.

Normalmente se necesitan cinco o seis intentos para conseguir un producto realmente adecuado.[66]
La mayor parte de las preocupaciones sobre el desarrollo iterativo implican el temor de que este enfoque dará lugar a una arquitectura o diseño ineficaz. Esto sucede porque las personas tienen la idea de que todo buen diseño ocurre al inicio del proyecto. Sin embargo, muchas personas que participan en el desarrollo de productos comprenden que los grandes diseños evolucionan con el tiempo. Cuanto más complejo es el sistema, mayor importancia adquiere la evolución del diseño.
Este pensamiento se refleja en las prácticas de producción Lean, donde la mejora continua es una estrategia clave. Nadie espera que un proceso de fabricación sea perfecto, ya que es demasiado complejo. En cambio, se espera que los trabajadores involucrados en la producción detengan la línea cuando las cosas no son perfectas, encuentren la raíz del problema y los solucionen antes de continuar con la fabricación. El sistema de producción Toyota comenzó con prácticas que se han mejorado continuamente por miles de trabajadores en las últimas décadas. Incluso hoy en día, se está mejorando este sistema de producción.
Se debe adoptar la actitud de que la estructura interna de un sistema requerirá la mejora continua a medida que el sistema evolucione. Así como los procesos de fabricación se mejoran continuamente por los trabajadores de producción, los sistemas de software deben ser mejorados continuamente por los desarrolladores. De hecho, los productos de software pasan por varias versiones, y por lo tanto se entiende que sin duda el producto ha sido mejorado en varias ocasiones. La refactorización, o sea la mejora del diseño a medida que el sistema se desarrolla, no es solo para el software comercial. Sin la mejora continua cualquier sistema de software se verá afectado. Las estructuras internas se volverán frágiles y en un tiempo sorprendentemente corto, el sistema dejará de ser útil.
La necesidad de refactorización surge a medida que la arquitectura evoluciona, madura y nuevas funciones son requeridas por los usuarios. Las nuevas funciones o características pueden ser agregadas al código una a una, pero en general están relacionadas entre sí, por lo tanto sería mejor añadir una mejora arquitectónica para apoyar el nuevo conjunto de características.[67]

5.6.5.1. MANTENIMIENTO DE LA INTEGRIDAD CONCEPTUAL

Todo sistema con integridad conceptual posee un conjunto de características principales que cuando comienzan a perderse, marcan la necesidad de refactorizar:[68]

Simplicidad: en casi todos los campos, un diseño sencillo y funcional es la mejor alternativa. Los desarrolladores experimentados entienden como simplificar código complejo, y de hecho, la mayoría de los modelos de desarrollo de software apuntan a aportar simplicidad a un sistema complejo.
Claridad: el código debe ser fácil de entender por todos aquellos que eventualmente trabajarán con él. Cada elemento debe ser nombrado de manera tal que comunique claramente lo que es o lo que hace sin necesidad de comentarios. Se deben establecer convenciones bien entendidas para nombrar los elementos y utilizar un lenguaje común. De este modo se obtendrá claridad en el código y una notación simple.
Adecuado para el uso: cada diseño debe cumplir con su propósito. Una interfaz de usuario que no es intuitiva no es adecuada. Cuando las pruebas muestran que el rendimiento se ha

degradado hasta un nivel inaceptable, la cuestión debe abordarse sin demora, incluso si esto significa cambiar el diseño.

Sin repeticiones: código idéntico nunca debe existir en dos o más lugares. La repetición indica un patrón emergente y debe dar una señal de alerta para clarificar el diseño. Cuando los cambios deben ser hechos en más de un lugar, la posibilidad de error crece de manera exponencial, por lo que la duplicación es uno de los mayores enemigos de la flexibilidad.[69]

Sin características adicionales: cuando un código ya no se necesita, el desperdicio implicado en el mantenimiento es grande. Esta pieza de código debe ser almacenada, compilada, integrada y probada cada vez que se realiza alguna modificación. Lo mejor es deshacerse de él. Lo mismo se aplica para las características o funciones creadas por precaución, anticipando posibles necesidades futuras. Anticipar el futuro suele ser inútil y consume recursos valiosos. Se puede tomar una opción en el futuro retrasando decisiones, pero no predecir el futuro, proporcionando características antes que sean necesarias.

Un buen diseño evoluciona a lo largo de la vida de un sistema, pero esto no sucede por accidente; un código pobre no mejora siendo ignorado. Cuando un desarrollador encuentra algún problema en el código base, lo que interfiere con la fluidez del desarrollo o la ejecución sin problemas, se debe detener el agregado de nuevas funciones. El equipo se debe tomar el tiempo necesario para encontrar y corregir la raíz del problema antes de continuar con el desarrollo. Una refactorización útil requiere un buen sentido del diseño. Equipos sin experiencia pueden llegar a cambiar el código varias veces sin mejorar el diseño, perdiendo tiempo al perfeccionar detalles sin importancia.

5.6.5.2. ¿REFACTORIZAR ES REPETIR TRABAJO?

La sabiduría popular sostiene que el desarrollo secuencial debería dar lugar a mejores productos con menos riesgo, mientras que el diseño y el desarrollo solapado darán lugar a la repetición de trabajo, siendo más costoso y consumiendo mucho más tiempo. Por lo contrario como se vio en el Capítulo 3 “Decidir lo más tarde posible”, el desarrollo simultáneo por lo general da mejores productos, más baratos, más rápidos y con menos riesgos. El desarrollo simultáneo significa que el diseño del producto emerge a lo largo del proceso de desarrollo. La mejora del diseño durante el desarrollo sin duda no representa la repetición de trabajo, por el contrario, es una buena práctica de diseño.

Se podría pensar que no hay tiempo para detener el desarrollo y mejorar el diseño, pero en realidad podríamos argumentar que no hay tiempo para no refactorizar. El trabajo solo irá más lento si el código se vuelve complejo y oscuro. Como sugiere la Figura 10.4, la falta de refactorización matará la productividad del equipo. Nadie en Toyota pensaría que parar una línea para encontrar y solucionar un problema volvería más lenta las cosas. Ellos saben que enfocarse en la mejora incesante hace que la línea vaya más rápido.

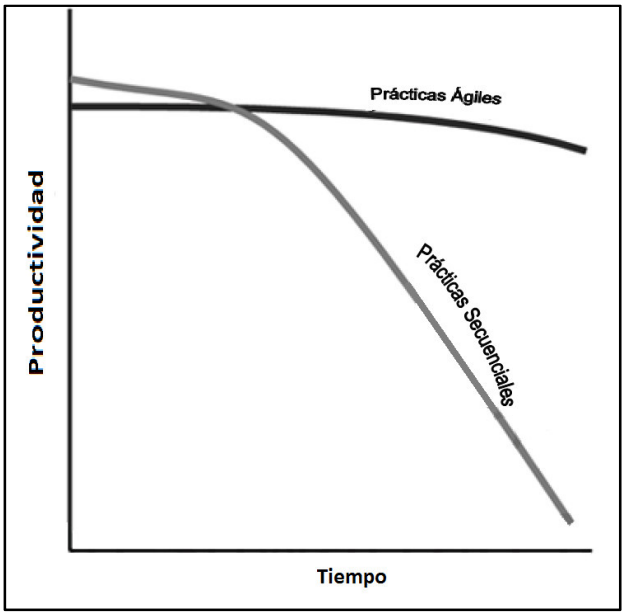


Figura 5.13 Mejorar continuamente el diseño sustenta la productividad

La refactorización no es desperdicio; por el contrario, es un método clave para evitar el derroche en la prestación de valor de negocio a los clientes. Un código base bien diseñado es el cimiento de un sistema que puede responder a las necesidades de los clientes, tanto durante el desarrollo como durante toda su vida útil.

5.6.6. PRUEBAS

En el desarrollo de software se prueba que el diseño se logre de manera correcta y que el sistema haga lo que los clientes quieren. Cuando los desarrolladores escriben el código, debe haber una prueba que asegure que cada función o característica trabaja como es debido y que todas las piezas funcionan en conjunto. Estas pruebas se han clasificado como pruebas de unidad, pruebas de sistema y pruebas de integración. A medida que se avanza de programar un módulo a la vez hasta programar capacidades y funciones completas, la distinción entre pruebas de unidad, de sistema y de integración tiene menos sentido. Una mejor denominación para estas pruebas podría ser “pruebas de desarrollo”, ya que su finalidad es garantizar que el código hace lo que el desarrollador pretende.
Las pruebas para asegurar que el sistema cumple correctamente con su objetivo han sido llamadas pruebas de aceptación, pero éste término se ha utilizado tradicionalmente para referirse a las pruebas que se ejecutan al final del desarrollo. Un nombre mejor para estas pruebas es “pruebas de clientes”, ya que su finalidad es garantizar que el sistema va a hacer lo que los clientes esperan que haga. Las pruebas de clientes se ejecutan durante todo el desarrollo y no solo al final.
Las pruebas desempeñan roles fundamentales durante el proceso de desarrollo de software. En primer lugar, las pruebas sin ambigüedades comunican cómo las cosas tienen que funcionar. En segundo lugar, proporcionan retroalimentación sobre si el sistema funciona realmente de la forma en que se supone que funcione. En tercer lugar, las pruebas otorgan el andamiaje que permite a los desarrolladores hacer cambios a lo largo del proceso de desarrollo, generando herramientas útiles en la práctica como el desarrollo basado en

conjuntos y la refactorización. Cuando el desarrollo se lleva a cabo, el conjunto de pruebas proporciona una representación exacta de cómo se construye el sistema. Por último, mediante el desarrollo y mantenimiento de conjuntos de pruebas de todos los sistemas de producción, se pueden realizar cambios de manera segura mediante la ejecución de una serie completa de pruebas para todas las aplicaciones relacionadas.

5.6.6.1. COMUNICACIÓN

Cuando un producto se lanza a la fabricación, también se liberan las pruebas para informar a la organización de fabricación lo que constituye exactamente un producto aceptable. De la misma manera, las pruebas de desarrollo transmiten exactamente como se supone que el sistema funcione internamente, mientras que las pruebas de clientes transmiten lo que ellos necesitan de una aplicación.
Las pruebas de clientes pueden ser un reemplazo viable o un complemento para la mayoría de los documentos de requerimientos. Supongamos que un desarrollador tiene una conversación con un cliente acerca de los detalles de una característica. La conversación no debe considerarse completa hasta que quede expresada como una prueba de cliente.
Ahora imaginemos una sesión de diseño rápida entre los desarrolladores que determinan como se ejecutará la función. La aplicación no estará completa hasta que los detalles del diseño se comprueben por medio de pruebas de desarrollo. Al documentar el diseño de las pruebas, los desarrolladores pueden escribir código con una comprensión clara de lo que se supone que debe hacer. Esta es una buena manera de refinar el pensamiento y ayudar a los desarrolladores a escribir código con integridad conceptual.
Existen alternativas a escribir las pruebas como un dispositivo de comunicación antes de la codificación, pero no hay alternativa a las pruebas de escritura para demostrar si el sistema hace lo que se supone que debe hacer. Por lo tanto se puede conseguir una doble función de las pruebas usándolas para documentar lo que se supone que el sistema debe hacer, al igual que la fabricación a menudo usa las pruebas para transmitir las especificaciones del producto.

5.6.6.2. RETROALIMENTACION

Cuando un desarrollador escribe código, debe recibir retroalimentación (feedback) inmediata sobre el funcionamiento del mismo en función de lo que se pretende. En otras palabras, debe haber una prueba para cada mecanismo implementado. En base a esto, se puede aprovechar el hecho de que el desarrollo es un ciclo de experimentos con una prueba de éxito al final de cada bucle.
La razón para desarrollar software en iteraciones cortas es que se puede proveer retroalimentación acerca de cómo funciona el sistema a los clientes o sus representantes y obtener su opinión sobre cómo proceder. Con el fin de conseguir ese feedback, se debe mostrar al cliente lo que el software desarrollado en la iteración hace por ellos. Dicho de otra manera, se necesita un conjunto de demos o scripts que demuestren la funcionalidad desarrollada. Estos deben ser entendidos por los clientes lo suficientemente bien como para asegurar que todo lo que les interesa haya sido implementado con éxito en la iteración. Puesto que se van a demostrar las características al final de cada iteración, es posible capturar la demostración en las pruebas realizadas y prescindir de la necesidad de tener testers.

En caso de tenerlos, las pruebas para desarrolladores y clientes deben ser automatizadas tanto como sea posible y ejecutadas como parte del trabajo diario.[70] Si no están automatizadas o toman demasiado tiempo, no se pueden ejecutar con la frecuencia suficiente. Se realizarán grandes lotes de cambios antes de las pruebas, lo que hará mucho más probable la aparición de fallas y más difícil detectar el cambio que las causó.

5.6.6.3. ANDAMIAJE

El andamiaje es una estructura de soporte que permite a los trabajadores hacer cosas que de otro modo serían peligrosas. Si se desarrolla software en iteraciones, se retrasan las decisiones hasta el último momento responsable y se usa desarrollo basado en conjuntos y refactorización, probablemente se realizarán serios cambios al código una vez escrito. Como se sabe, esto es peligroso ya que los cambios tienden a tener consecuencias no deseadas. Cualquier sistema no trivial requiere que cientos de miles de detalles sean correctos al mismo tiempo, en muchos casos interactuando de manera demasiado compleja para anticipar. Para realizar cambios de forma segura, debe haber una manera inmediata de encontrar y corregir las consecuencias no deseadas. La manera más eficaz es tener un conjunto de pruebas automatizadas que verifiquen los mecanismos que los desarrolladores intentan implementar y el comportamiento que los clientes requieren. Un buen conjunto de pruebas encontrará consecuencias no deseadas de inmediato y determinará la causa del problema.
En este sentido, los conjuntos de pruebas automatizadas son el andamiaje que proporciona la seguridad y el acceso a los constructores del sistema de software a medida que completan la construcción de un edificio parcialmente terminado. No se puede utilizar con eficacia las herramientas vistas en éste capítulo sin ese andamiaje. Puede parecer que escribir pruebas ralentiza el desarrollo, pero en realidad las pruebas valen la pena, tanto durante el desarrollo como durante el ciclo de vida del sistema.
Cuando se piensa en ello, las pruebas son fáciles de encontrar, formalizar y automatizar, ya que los desarrolladores comprueban su trabajo de alguna manera a medida que codifican y encuentran la manera de demostrar a los clientes como funciona el sistema al final de cada iteración. Lo que se debe hacer es capturar esas pruebas, asegurarse de que sean correctas y completas, automatizarlas, considerarlas como parte del producto liberado y continuar utilizándolas y mejorándolas. Se podría terminar con tantas líneas de código de prueba como de producto, pero los beneficios siempre superarán el costo.

5.6.6.4. DOCUMENTACION CONFORME A OBRA (AS-BUILT)

No es para nada sorpresa el hecho de que mantener precisa la documentación as-built de un software es difícil o imposible. Se han realizado muchos intentos para concretar esto, pero siempre se detectaron fallas. Sin embargo, si un sistema tiene un conjunto de pruebas integral que contenga pruebas de desarrolladores y clientes, las mismas de hecho serán un reflejo más preciso de la construcción del sistema. Si las pruebas son claras y bien organizadas, son un recurso muy valioso para la comprensión del funcionamiento del sistema desde el punto de vista del desarrollador y el cliente.
La otra cosa que un conjunto de pruebas hace es dar una indicación de la salud del sistema integrado. El recuento de defectos, tipos y tendencias son una muy buena indicación de la convergencia, del momento propicio para su lanzamiento y la solidez del sistema.

La conclusión es que se debe tener (en la medida que sea práctico) un conjunto de pruebas de desarrollo y clientes completo y automatizado. Deben estar sujetas a la misma disciplina en el diseño, semántica, versionado, construcción, sincronización y refactorización como el propio sistema. Si no parece haber suficiente tiempo, lo primero que se debe hacer es reasignar el esfuerzo empleado en la documentación de requisitos a escribir pruebas de clientes. Se debe asegurar que los desarrolladores escriban y automaticen sus propias pruebas de desarrollo, mientras se provee capacitación y entrenamiento en el desarrollo y automatización de pruebas. Se puede obtener más beneficios de un programa de prueba eficaz que de la mayoría de las inversiones posibles de realizar.

5.6.6.5. MANTENIMIENTO

La industria del software necesita encontrar una manera de facilitar los cambios en el sistema una vez lanzado a la producción, ya que más de la mitad del desarrollo se produce luego de la liberación inicial. Además, realizar los cambios debe ser económico, o sea con rapidez y a un costo razonable. Hay muchas maneras de hacer al software más amigable al cambio: estratificación, agrupamiento, ocultar la variabilidad potencial, componentes, uso de software comercial, etc. También es una buena alternativa mantener al equipo de desarrollo responsable del mantenimiento de la aplicación para preservar el aprendizaje del dominio. Todas estas técnicas son importantes, pero se debe añadir otro mecanismo a la lista: el mantenimiento de una serie de pruebas exhaustivas en todo el ciclo de vida del sistema. Si un andamiaje de pruebas fue construido durante el desarrollo, todo lo que se debe hacer es volver a erigir el andamio y proceder con los cambios. Entonces, el sistema puede ser reparado y rediseñado a lo largo de su vida útil de forma segura. El andamiaje es tan útil para el mantenimiento como lo fue para la construcción original.
Supongamos que se tiene un sistema complejo con muchas aplicaciones utilizando servicios comunes (una base de datos común, middleware o hardware, por ejemplo). Se puede sospechar que un cambio en una sola aplicación podría tener una reacción adversa en una aplicación relacionada. Como es muy difícil poseer un conjunto fiable de documentación as-built, se debe averiguar la manera en que todos los sistemas funcionan realmente antes de poder cambiar de forma segura a cualquiera de ellos. No es de extrañar que el mantenimiento sea dificultoso.
Lo que se necesita es el conjunto de pruebas para cada aplicación, desarrollada como andamiaje para el cambio durante el desarrollo. Estas pruebas constituyen un conjunto preciso de documentación as-built para todas las aplicaciones del entorno. Si cada aplicación tiene un conjunto actualizado de pruebas para demostrar su integridad, se puede probar todo el entorno antes de implementar el cambio.

5.7. SEPTIMO PRINCIPIO - VER TODO EL CONJUNTO

5.7.1. PENSAMIENTO SISTEMICO

Un sistema está formado por partes interdependientes que interactúan unidas por un propósito. Un sistema no es sólo la suma de sus partes, es el producto de sus interacciones. Las mejores partes no hacen necesariamente el mejor sistema, la capacidad de un sistema para lograr su propósito depende de qué tan bien las partes trabajen juntas, no sólo el nivel de rendimiento que tienen individualmente.
El pensamiento sistémico ve a las organizaciones como sistemas, analiza cómo las partes de una organización se interrelacionan y se desempeñan en su conjunto a través del tiempo. Cuando éste análisis se lleva a cabo mediante la construcción de una simulación del comportamiento de la organización, se la denomina dinámica del sistema. Los analistas de la dinámica de sistemas construyen un modelo computarizado mediante entrevistas a las personas para descubrir la política operativa de la organización para la toma de decisiones y los bucles de retroalimentación dentro de la misma. Los analistas generalmente encuentran un amplio acuerdo dentro de una organización sobre cómo se toman las decisiones y descubren que la mayoría de las personas toman decisiones coherentes basadas en datos apropiados. Sin embargo, la simulación revela por lo general sorprendentes consecuencias no deseadas de las políticas aparentemente correctas, señalando que el mayor impacto de las políticas locales no es bien comprendido.
El especialista en dinámica de sistemas JayForrester señala que un modelo computarizado basado en las políticas conocidas en una empresa a menudo predice las mismas dificultades que la compañía ha estado experimentando. Señala que las políticas establecidas para resolver un problema con frecuencia lo exacerba, creando una espiral descendente: a medida que el problema empeora, los gerentes aplican incluso más agresivamente las políticas que están causando el problema.[71]
A menudo vemos esta dinámica en el desarrollo de software. Cuando una organización experimenta problemas de desarrollo, hay una tendencia a imponer un proceso más “disciplinado” en la organización, por lo general uno con el procesamiento secuencial más riguroso: documentación de requisitos más completas, obtener la aprobación del cliente por escrito, controlar cambios con más cuidado y trazar cada requerimiento en el código. Si una organización carece de disciplinas básicas de desarrollo, la imposición de un proceso secuencial riguroso inicialmente puede mejorar la situación. El pensamiento sistémico advierte que el hecho de que las cosas mejoren no significa que la solución es la correcta. Los efectos tardíos de un proceso secuencial en un entorno que evoluciona con el tiempo finalmente tendrán consecuencias; cada vez será más difícil mantener el sistema en función de las necesidades de los clientes. En ese momento, forzar un proceso secuencial aún más riguroso iniciará una espiral descendente.
Uno de los modelos básicos de pensamiento sistémico se denomina *límites de crecimiento*.[72] Incluso si un proceso produce el resultado deseado, se crea un efecto secundario que equilibra y eventualmente retrasa el éxito. Si se continúa insistiendo en el mismo proceso para incrementar el éxito, se amplificará el efecto secundario y se iniciará una espiral descendente. En vez de forzar el crecimiento, se debe encontrar y eliminar sus límites.

Encontrar y eliminar los límites de crecimiento es la enseñanza fundamental de la teoría de las restricciones.[73] La idea es buscar y eliminar la actual restricción de crecimiento, reconociendo que ésta se trasladará a otro lugar una vez que sea abordada, por lo que esto es un proceso continuo. De hecho, las políticas del pasado pueden convertirse en limitaciones actuales.[74]

Un segundo modelo básico en el pensamiento sistémico se llama *desplazamiento de la carga*.[75] En este modelo un problema subyacente produce síntomas que no pueden ser ignorados. Sin embargo, el problema de fondo es difícil de enfrentar y las personas afrontan los síntomas en lugar de la causa raíz del problema. Lamentablemente, la solución rápida permite que el problema de fondo crezca inadvertidamente ya que sus síntomas se han ocultado.

El pensamiento Lean utiliza *cinco* "*por qué*"[76] para contrarrestar la tendencia de trasladar la carga a los síntomas en lugar de abordar la causa raíz de un problema. Los cinco "*por qué"* funcionan de la siguiente manera: supongamos que se tiene un problema con un número creciente de defectos. Al preguntarse *por qué* se producen los defectos, se descubre que un nuevo módulo se ha añadido y tiene consecuencias imprevistas. Luego, la interrogante que se presenta es *por qué* el nuevo módulo genera defectos en los otros módulos, descubriendo que no se ha probado. El *porqué* de esto es que los desarrolladores trabajaron bajo presión para entregarlo antes de que se probara. El *porqué de* dicha presión reside en el pensamiento de que los desarrolladores trabajan mejor con plazos estrictos, por lo que un plazo artificial fue forzado. Pero aún el problema no ha concluido. Se tiene por lo menos una pregunta más que responder antes de llegar a la raíz del problema. Al preguntarse *por qué* alguien pensó que los plazos artificiales eran necesarios, se descubre que el gerente tiene un miedo intenso a no cumplir con el cronograma de desarrollo. Por lo tanto, se pierde tiempo explicando a este gerente cómo funciona el cronograma de trabajos pendientes y cómo de acuerdo a éste el sistema convergería, fundamentando cómo la presión de un cronograma poco razonable aumenta los defectos y prolonga los tiempos de trabajo.

Un tercer modelo básico en el pensamiento sistémico es la *sub optimización*. Cuanto más complejo es un sistema, más tentación hay de dividirlo en partes y gestionarlas a nivel local. La gestión local tiende a crear mediciones locales de rendimiento, las que a menudo crean efectos en todo el sistema que disminuyen el rendimiento general, sin embargo; el impacto de la optimización local sobre los resultados globales a menudo se oculta.

5.7.2. MEDICIONES

Por lo general se tiene una tendencia a dividir un gran trabajo en tareas más pequeñas. Entonces, parece elemental que la manera de obtener el mejor resultado global es la optimización de los resultados de cada trabajo individual, porque se cree que si se consigue mejores mediciones en cada tarea, éstas se sumarán y brindaran una mejor medición global del trabajo. Pero la optimización de todas las tareas es a menudo una muy mala estrategia.

5.7.2.1. OPTIMIZACION LOCAL

Nos encontramos con un departamento de pruebas que es medido en función de la proporción aplicada de los testers, es decir, el porcentaje de las horas de prueba disponibles que se recargan a otros departamentos. Para aumentar los números de la proporción aplicada, el gerente del departamento mantiene bajo el número de testers y deja apilar una gran cumulo de trabajo delante de cada uno para asegurarse de que todo el personal tenga mucho trabajo que hacer. Los departamentos de clientes con sistemas para probar tienen que esperar mucho tiempo para realizar la prueba, lo que aumenta su tiempo de ciclo y reduce su retroalimentación, lo que generalmente se traduce en productos de menor calidad con más defectos. Esto eleva la cantidad de pruebas necesarias y la carga de trabajo del departamento de pruebas.
Enfocarse en la relación aplicada en el proceso de prueba es lo mismo que enfocarse en la utilización de la maquinas en el proceso de manufactura. Aunque los gerentes de fabricación y sus contadores han aprendido que ésta es una medida de sub optimización, el departamento de prueba en cuestión y sus contadores todavía deben adoptar ésta idea.
Uno de los problemas más difíciles en la medición de los resultados es que éstas se producen a nivel local, pero maximizar las mediciones locales a menudo va en contraposición a la optimización de la organización en su conjunto. Sin embargo, no siempre es evidente que la optimización local está perjudicando a toda la organización, como en el caso del departamento de pruebas. Los costos de acumular inventario (si se trata de trabajo a la espera de ser realizado o características adicionales añadidas por si acaso) están ocultas del sistema de medición. El beneficio económico de un flujo de valor rápido a través de la cadena de valor también es difícil de medir, pero el aumento del flujo es una excelente manera de identificar y eliminar los residuos generados por medidas de sub optimización.
Enfocarse exclusivamente en las mediciones locales crea una tendencia a inhibir la colaboración más allá del área que se está midiendo, ya que no hay ninguna recompensa por ello. Tomando nuevamente el departamento de pruebas, los testers no fueron medidos en su capacidad de colaborar con los desarrolladores y ayudar a disminuir los defectos, por lo que probablemente no se involucraron en la mejora del proceso de desarrollo. Como se vio en el capítulo 5 “Potenciar el equipo”, ésta cuestión se aborda basando los incentivos en mediciones de un nivel más alto al que uno esperaría. Si aplicamos el mismo concepto en este caso, los desarrolladores y los testers serían reconocidos conjuntamente por una tasa de defectos baja, dándoles más incentivos para colaborar.

5.7.2.2. ¿POR QUE SE SUB OPTIMIZA?

Los efectos perjudiciales de las mediciones locales en el rendimiento global se encuentran ocultos generalmente, por lo que persisten en el uso de mediciones de sub optimización por superstición y hábito.[77]

5.7.2.3. SUPERSTICION

La superstición es una relación sin fundamento de causa y efecto. Algunas supersticiones son inofensivas, por ejemplo, usar una camisa roja para que un equipo gane. De hecho, cada vez que se usa esa camisa roja el equipo gana, y cuando no se usa, pierde. Se sabe que esa camisa roja no está causando que el equipo gane, pero es agradable pensar que sí.
Otras supersticiones son más perjudiciales, por ejemplo, creer que cuando la proporción aplicada de los testers es alta, ésta repercute directamente en los resultados finales y los

beneficios aumentan. Cuando la relación aplicada sube aún más y los beneficios caen, se atribuye la caída a otra cosa. Se tiene la superstición de que una alta proporción aplicada significa grandes ganancias.

5.7.2.4. HABITO

El jefe del departamento de pruebas puede optimizar la relación aplicada por hábito, esa es la forma en que el departamento siempre ha trabajado. Las típicas mediciones de costos y control de cronograma del proyecto se hacen a menudo también por hábito. Se puede no creer que son realmente las mediciones más importantes para el éxito del proyecto, pero como todos la realizan se cree que deben ser importantes.

5.7.2.5. MEDICION DEL RENDIMIENTO

"Cuando se trata de medir el rendimiento, sobre todo el desempeño de los trabajadores del conocimiento, se está *cortejando* positivamente disfunción". Particularmente éstas son palabras fuertes de Tom DeMarco y Timothy Lister en el prólogo del libro de Rob Austin "*Measuring and Managing Performance in Organizations"*.
La teoría de Austin tiene mucho sentido.[78] Su premisa es que las personas tratarán de optimizar las mediciones en las que se basarán su desempeño. El problema es que es muy difícil medir todo lo que es importante en el trabajo de conocimiento, sobre todo cuando cada esfuerzo es único y reina la incertidumbre. Se mide lo que se puede y se espera hacer que suficientes cosas funcionen bien para obtener los resultados generales que se desean.
Esto no es exactamente así. La regla básica "se obtiene lo que se mide" aún se mantiene. Si no se puede medir todo lo que es importante, las mediciones parciales son muy propensas a convertirse en mediciones sub optimizadas. Si no se puede medir todo lo que es necesario para optimizar el objetivo general de la empresa, entonces se está mejor sin las mediciones parciales de sub optimización. De lo contrario se está en serio peligro de fomentar una conducta de sub optimización.
Nuestra cultura es adversa a esta conclusión; las mediciones de desempeño parecen fundamentales para la manera en que hacemos negocios. Austin señala que la mayoría de los gerentes quieren utilizar medidas de rendimiento y tratan de crear medidas que abarcan todo. Esto se hace de tres maneras:[79]

Estandarizar: se abstrae en el proceso de desarrollo en fases secuenciales y se crean estándares sobre cómo se debe hacer cada fase. Luego se mide la conformidad con el proceso.
Especificar: se crea una especificación detallada o plan, se mide el desempeño en base al plan y se encuentra la variación en función a éste.
Descomponer: se divide las grandes tareas en otras más pequeñas y se mide cada tarea de manera individual.

Si Austin tiene razón en esto, las prácticas tradicionales de gestión de desarrollo de software provienen de un deseo de medir el trabajo complejo no estructurado mediante desagregaciones. Lamentablemente estas medidas probablemente fomentarán el comportamiento de sub optimización debido a que todavía no se mide todo lo que es importante. La manera de asegurarse de que todo es medido es mediante agregación, y no

por desagregación. Es decir, mover la medición un nivel hacia arriba y no un nivel hacia abajo.

5.7.2.6. MEDICIONES DE INFORMACION

Las mediciones son importantes para el seguimiento del progreso del desarrollo de software. Por ejemplo, el recuento de defectos es muy importante para estimar si el software se encuentra listo para su lanzamiento. Sin embargo, las mediciones de información, y no las de desempeño, deben ser utilizadas para éste propósito. Las mediciones de información se obtienen agregando datos para ocultar el desempeño individual. Un sistema de medición de defectos es un sistema de medición del rendimiento si se atribuyen los defectos a las personas y se convierte en un sistema de información si se agrega defectos por función. Austin es bastante explícito al afirmar que es importante agregar mediciones de desempeño en lugar de atribuírselas a los individuos.[80]
El problema con la atribución de los defectos a los desarrolladores radica en el supuesto de que los individuos causan los defectos personalmente. Antes se pensaba que los trabajadores de las fábricas causaban personalmente los defectos de calidad y que si fueran más cuidadosos, habría menos defectos. Luego se aprendió del movimiento de calidad en la década de 1980 que menos del 20 por ciento de todos los defectos de calidad están bajo el control de los trabajadores. El resto tienen sus raíces en los sistemas y procedimientos vigentes, que están bajo el control de la gestión y no el control de los trabajadores.[81]
Se puede decir que la misma idea es cierta en la mayoría de los entornos de desarrollo de software: La gran mayoría de los defectos tienen su raíz en los sistemas y procedimientos de desarrollo y tratar de atribuir los defectos a los desarrolladores individuales es un caso de trasladar la carga de responsabilidad. La manera de encontrar la causa raíz de los defectos es animar a toda la organización de desarrollo a colaborar en la búsqueda y eliminación. Atribuir los defectos a las personas desalienta este tipo de colaboración, mientras que agregarlos dentro de las medidas de información no apunta a los individuos y ayuda en la búsqueda de la causa de los problemas.

5.7.3. CONTRATOS

5.7.3.1. CONFIANZA ENTRE EMPRESAS

A menudo se presenta la interrogante de cómo aplicar el desarrollo ágil cuando se debe trabajar bajo contrato. Sin duda el mayor obstáculo para su uso es la línea divisoria entre una empresa y otra. Cada empresa tendrá en cuenta sus propios intereses, con el entendimiento de que la otra empresa hará lo mismo. Parecería entonces que el único método seguro consiste en escribir un contrato estricto, porque las personas migran hacia nuevos puestos de trabajo, las reglas cambian y entonces lo único que importa es lo que está plasmado en el contrato.
De hecho existe una mejor manera que fue promovida por Toyota cuando comenzó a trabajar con proveedores de Estados Unidos en 1988 y fue documentada por Jeffrey Dyer en CollaborativeAdvantage. Desde luego, Toyota negoció contratos con sus proveedores, pero los contratos no fueron los principales medios que protegían los intereses de los proveedores. En un tiempo sorprendentemente corto, los proveedores desarrollaron

confianza en Toyota, y en 1998, fue calificada por los proveedores de automóviles como el fabricante más confiable del país, con ventas dos veces superior a General Motors.[82] La *confianza* en este caso tiene un significado específico:

La medida en la que se puede confiar en el fabricante de automóviles para tratar a un proveedor de manera justa.
Hasta qué punto el fabricante de automóviles podría intentar aprovecharse indebidamente del proveedor.
La reputación de imparcialidad del fabricante automotriz en la comunidad de los proveedores.

Este tipo de confianza no proviene de personas confiando entre sí. Los proveedores pueden confiar completamente en un agente de compras individual, pero no pueden confiar en que la misma persona estará allí un año más tarde o que todavía se estará jugando bajo el mismo conjunto de reglas. Los proveedores desarrollaron "mayor confianza en la equidad, estabilidad y previsibilidad en las rutinas y procesos de Toyota".[83]
Dyer señala que los proveedores comparten información confidencial con Toyota, confiando de que no será revelada a sus competidores, como a veces ocurrió con General Motors. Los proveedores invirtieron en equipos especializados, sabiendo que Toyota repite los negocios con sus proveedores el 90% de las veces, mientras que solo tenían el 50% de probabilidad de repetirlos con GM. Los proveedores permitieron que expertos de Toyota enseñaran el sistema de producción Toyota en sus plantas, comprendiendo que Toyota no exigiría reducciones de precios basadas en sus resultados, como GM supo hacer.[84]
No se debe pensar que Toyota no mira sus propios intereses; simplemente entiende que una red de proveedores fuerte es mucho más beneficiosa para sus intereses que las ganancias a corto plazo resultante de tomar ventaja de un proveedor. En los Estados Unidos Toyota se obtiene de proveedores cerca de las tres cuartas partes de sus componentes, mientras que los fabricantes de automóviles de Estados Unidos obtienen menos de la mitad de sus componentes de los proveedores. Sin embargo, Toyota gasta la mitad de dinero y tiempo en materia de contratación que GM. Además, los proveedores son más productivos y producen mejor calidad en células de fabricación dedicadas a Toyota.[85] Como organización, está muy consciente que las relaciones de colaboración sirven mejor a sus intereses que las relaciones normales de mercado con la mayor parte de sus proveedores.

5.7.3.2. CONFIANZA EN EL DESARROLLO DE SOFTWARE

Se podría decir que las buenas relaciones con los proveedores son importantes cuando están desarrollando algo que se puede fabricar muchas veces, como una unidad de disco y una luz trasera. Pero en el caso del software, se desarrolla un sistema una sola vez. Éste es complejo, costoso, está sujeto a muchos cambios y si no se hace bien, el impacto financiero puede ser tremendo. Entonces: ¿Dónde está el equivalente a esto en la fabricación?
Al comienzo del desarrollo del tercer principio Lean, "Decidir lo más tarde posible", se discute sobre las grandes y costosas matrices utilizadas para fabricar paneles de las carrocerías de vehículos; el costo de esas matrices representa cerca de la mitad de inversión de capital de un nuevo modelo. Son complejas, costosas y sujetas a muchos cambios, incluso después de que el diseño está supuestamente congelado. La corrección de un error cometido en el corte de la matriz consume mucho tiempo y el costo de empezar de

nuevo es alto. Sin embargo, a finales de la década de 1980, Toyota desarrolló matrices a la mitad del costo y en tan solo la mitad del tiempo usando prácticas de desarrollo simultáneo, en comparación con la típica empresa americana que utiliza desarrollo secuencial. Por otra parte, las matrices resultantes dieron a Toyota una ventaja de costos significativos en el proceso de fabricación.[86]

Los fabricantes de herramientas y matrices son empresas proveedoras tanto en Estados Unidos y Japón. Los fabricantes de automóviles americanos esperaron a que se congelaran las especificaciones de diseño y luego enviaron el diseño final al proveedor de matrices, lo que desencadenó el proceso de ordenar el bloque de acero y cortarlo. Los cambios tenían que ser aprobados y oficialmente enviados al proveedor por el departamento de compras. Dado que los proveedores tenían que hacer una oferta mínima para conseguir el trabajo, ellos obtuvieron la mayor parte de sus ganancias a partir de las órdenes de cambio, ascendiendo de un 30 por ciento a un 50 por ciento del costo de la matriz.[87]

En Japón, los proveedores de herramientas y matrices comienzan a trabajar en una matriz al mismo tiempo en que el diseño del automóvil inicia. Se espera que el fabricante sepa la implicancia de realizar una matriz para una parte y está en constante comunicación con el diseñador. Supongamos que un cuerpo de ingenieros quiere realizar un cambio; van directamente a la tienda de matrices, analizan la modificación propuesta con los ingenieros de matrices, ellos comprueban la factibilidad de producción y juntos deciden que acción llevar a cabo. El taller de matrices hace los cambios en la máquina fresadora y continúa con el corte de la matriz. Luego los trámites y aprobaciones siguen adelante.[88]

En Toyota, los contratos de herramientas y matrices son contratos de objetivos de costos, el proveedor y el fabricante de automóviles acuerdan el objetivo de costo total de las herramientas, incluyendo todos los cambios. Normalmente, los cambios suman al costo base un 10 o 20 por ciento más y esto está contemplado en el contrato original. Si el costo objetivo no se puede cumplir, las partes negocian sobre quien recae el costo adicional y, en general, Toyota termina con la mayor parte. Este tipo de acuerdo da a los ingenieros de ambas compañías el incentivo para trabajar en conjunto para mantener los costos dentro de la meta.

En los Estados Unidos, los fabricantes de herramientas tenían contratos de precio fijo que apuntaban a las ofertas más bajas, de modo que se vieron los cambios de ingeniería como oportunidades de lucro.[89] Para contener los costos, los fabricantes de automóviles pusieron en marcha un riguroso proceso de aprobación de cambio, igual a los que se encuentran en muchos contratos de desarrollo de software. Cuando se analiza el resultado global, el enfoque de EE.UU. casi duplicó el costo y el tiempo necesario para hacer una matriz. Por otra parte, dio como resultado una matriz de calidad inferior.[90]

El impacto general de muchas de las políticas de contratación y de control de alcance en el desarrollo de software está al mismo nivel. Es decir que un contrato de precio fijo con un vendedor esperanzado de sacar provecho de los cambios, en combinación con los rigurosos mecanismos de aprobación de cambio para contener los costos, puede aproximadamente doblar el costo y el tiempo que se necesita para desarrollar el software, mientras se producen resultados de menor calidad.

5.7.3.3. EL PROPOSITO DE LOS CONTRATOS

Dyer define la confianza como “La seguridad de una de las partes de que la otra cumplirá sus promesas y no explotará sus vulnerabilidades”.[91] Mucha gente piensa que la razón de

realizar contratos es para sustituir ésta confianza. La sabiduría popular dice que todas las eventualidades deben especificarse en un contrato para que las partes no puedan tomar ventaja una de otra.

A muchas empresas les resulta casi imposible seleccionar proveedores utilizando un proceso que valore la buena fe o escribir contratos que asuman que la otra parte actuará de buena fe. Es muy frecuente que la finalidad de los contratos sea limitar la tendencia natural de una de las partes para tomar ventaja de la otra, ya que se vela por los propios intereses.[92] Sin embargo, si el comportamiento perjudicial puede limitarse a través de las relaciones y no del contrato, puede dar como resultado toda clase de ventajas en términos de velocidad, flexibilidad, costo e intercambio de información.

Si se analiza por qué las empresas trabajan con proveedores, vemos que a medida que nuestro mundo se vuelve más complejo se valora cada vez más la especialización. Si Dell quiere la mejor placa de vídeo, colabora con la empresa que la fabrica. Si se desea el mejor software para un área en particular, es probable que se busque empresas que sean expertas en proporcionar ese tipo de software.

Otra de las razones para subcontratar el desarrollo de software es reducir los costos y mejorar la probabilidad de éxito. Por ejemplo, una organización puede encontrar que el sueldo de un vendedor es inferior a sus propios salarios. O podría negociar un precio fijo para un sistema que es más bajo que el costo interno para hacer el mismo trabajo. Puede descubrir que un proveedor de desarrollo de software con experiencia tiene un conjunto de habilidades que no están disponibles internamente.

Si examinamos los costos de la ecuación, vemos que el dinero pagado a los vendedores es solo una parte de la historia. Adicionalmente, existen costos de transacción, como por ejemplo el costo de la selección de posibles proveedores, negociación y renegociación de los acuerdos, monitoreo y cumplimiento del acuerdo, facturación y seguimiento de los pagos. Como se demostró en el ejemplo de la fabricación de matrices, el costo de tratar de controlar los cambios puede añadir más costos ocultos al contrato y se puede esperar que aumenten en un dominio que evoluciona.

Dyer considera que los costos de transacción dominan la mayoría de las relaciones entre vendedores y proveedores.[93] Por lo tanto, al evaluar el costo de la contratación externa, es imperativo que todos los gastos sean considerados: costos directos, costos de transacción y costos ocultos que provienen de las relaciones entre partes independientes e intolerancia al cambio. Estos costos serán especialmente altos en un ambiente que va a cambiar a pesar de los esfuerzos por mantener el cambio controlado.

Otro costo de la subcontratación es el de la oportunidad perdida que puede surgir si la comunicación entre el cliente y el vendedor es limitada. La integridad del sistema depende de la comunicación amplia, temprana y frecuente entre el cliente y el desarrollador. La falta de comunicación es una causa frecuente de fallo del sistema.[94] Teniendo esto en cuenta la probabilidad del aumento del éxito es una razón para la tercerización.

Los contratos que se enfocan en cuidar a las partes para que no tomen ventaja una de otra tienen una gran cantidad de mecanismos de control incorporados y puertas de comunicación que tienen una tendencia a aumentar los costos y reducir la colaboración fundamental para el éxito. Los contratos que se centran en el apoyo a la colaboración son más propensos a reducir costos y ser exitosos.

5.7.3.3.1. CONTRATOS DE PRECIO FIJO

Éste es el contrato diseñado para proteger al cliente de uso más común. A veces los ciclos presupuestarios corporativos y los procesos relacionados requieren contratos de precio fijo. Para muchas entidades del gobierno, la ley exige que los contratos de precio fijo a menudo sean adjudicados a la oferta más baja. Como hemos visto en el ejemplo del corte de matrices, ésta práctica alienta a vendedores a hacer ofertas bajas y obtener sus ganancias en los cambios. Otra motivación para éste tipo de contrato es el deseo de un cliente para transferir el riesgo al vendedor. En la práctica, el cliente no puede realmente trasladar la mayor parte del riesgo; si el contrato no funciona el cliente se verá afectado.

Como se señala en el desarrollo del segundo principio Lean, "Amplificar el Aprendizaje", es una buena idea desarrollar software en iteraciones cortas impulsadas por las necesidades inmediatas del cliente, desarrollando primero características de alta prioridad y deteniéndose cuando los recursos se agotan. Sin embargo, éste enfoque es muy arriesgado para los vendedores que trabajan con contratos de precio fijo, ya que con frecuencia tienen dificultades para acordar con el cliente que el trabajo se terminará cuando el dinero se acabe. Por lo tanto, los vendedores tienden a protegerse a sí mismos creando una especificación detallada y manteniéndola bajo un estricto control de cambio, facturando adicionalmente por cualquier modificación. El resultado puede ser un aumento sustancial del costo o un cliente muy decepcionado.

Los riesgos deberían surgir por la parte que mejor pueda tratarlo, y en un contrato de precio fijo el riesgo se transfiere aparentemente al vendedor. Si un problema es técnicamente complejo, el vendedor está más probablemente en condiciones de gestionar el riesgo asociado, por lo que es apropiado que éste asuma el riesgo. Sin embargo, si un problema es incierto o cambiante, el cliente se encuentra en la mejor posición para gestionar el riesgo, por lo que un contrato de precio fijo debería evitarse. Si un contrato de precio fijo no se puede evitar, entonces el cliente debe estar dispuesto a aceptar un costo sustancial más allá del precio fijo debido a la certeza de los cambios.

El contrato de precio fijo puede implicar un riesgo significativo en la estimación de los costos antes de realizar cualquier trabajo. Un vendedor competente incluirá éste riesgo en la oferta, mientras que uno que no entienda la complejidad del problema probablemente rebajará sus ofertas. El proceso de selección de un vendedor para estos contratos tiene la tendencia a favorecer al más optimista o al más desesperado.[95] En consecuencia, el vendedor con menos probabilidades de comprender la complejidad del proyecto tal vez sea el seleccionado. Es así que los contratos de precio fijo tienden a elegir al vendedor mar propenso a tener problemas.

Es muy común para el cliente encontrar un vendedor incapaz de entregar bajo un contrato de precio fijo. Al momento en que esto se hace evidente, el cliente rara vez tiene la opción de elegir otro vendedor, por lo que a menudo tiene que intervenir para salvar la situación. Como alternativa, el vendedor puede intentar recuperar su pérdida a través de órdenes de cambio, lo que lleva al cliente a evitar de forma agresiva cualquier modificación en el contrato. Frente a ninguna otra forma de recuperar una pérdida, el vendedor se verá motivado a encontrar formas de ofrecer menos de lo que el cliente realmente quiere. Un contrato de precio fijo está sesgado a favor del cliente a expensas del vendedor, por lo que es necesario para éste proteger sus intereses a costa del cliente. Por lo tanto se genera un clima en el que la confianza organizacional no tiene muchas posibilidades de crecer.

5.7.3.3.2. CONTRATOS DE TIEMPO Y MATERIALES

El contrato de precio flexible, también conocido como de *tiempo y materiales* o de tiempo y gastos, está diseñado para hacer frente a la incertidumbre y la complejidad, pero no elimina el riesgo, sino que simplemente lo desplaza desde el vendedor al cliente. En la década de 1970, el Departamento de Defensa de EE.UU. experimentó algunos rescates financieros de un perfil muy alto en contratos de precio fijo, por lo que comenzó a utilizar con más frecuencia contratos de tiempo y materiales en situaciones en que el gobierno estaba en mejores condiciones para gestionar el riesgo.
La desventaja desde el punto de vista del vendedor, es que los contratos de tiempo y materiales ofrecen menos seguridad que los de precio fijo. Sin embargo, estos contratos se consideran por lo general un buen negocio para los vendedores durante el tiempo que duren. De hecho, los vendedores generalmente tienen pocos incentivos para ser eficientes, porque cuanto más tiempo les toma el trabajo, más dinero ganarán. Para evitar este comportamiento por parte de los vendedores de materiales y tiempo, el Departamento de Defensa desarrolló amplios mecanismos de control para el vendedor, lo que contribuyó al desarrollo de la disciplina de gestión de proyectos.
Los contratos de tiempo y materiales marcan un aumento significativo en los costos de transacción. Las compañías con contratos del Departamento de Defensa no sólo contratan a los administradores para supervisar el cumplimiento de los requerimientos del contrato, sino que también agregan contadores para resolver los costos autorizados y no autorizados. Los elevados costos de transacción serían razonables si agregaran valor, pero en realidad estos costos por definición no agregan. Los controles no aportan nada de valor positivo, su único propósito consiste en ayudar a evitar residuos. En la medida en que estos hacen lo que se supone que deben hacer, pueden generar ahorros sustanciales, pero se debe reconocer que los controles son a su vez muy costosos.
Una forma de evitar el alto costo de los controles es no utilizarlos. Cuando los gastos de control son elevados, lo mejor sería mantener el trabajo dentro de una organización vertical, donde se presume que la administración controlará el comportamiento oportunista. Por desgracia, la integración vertical no siempre funciona para reducir al mínimo los costos de control, de hecho muchas organizaciones se encuentran utilizando internamente los controles de gestión de proyectos al estilo del Departamento de Defensa. Parece incongruente que ese costo, cronograma y mecanismos de control de alcance que aumentan el costo pero no el valor y que se inventaron para impedir que las partes contractuales se aprovechen uno del otro llegarían a dominar el desarrollo dentro de las empresas (el mismo lugar en el que no deberían ser necesarios).
Los contratos de tiempo y materiales pueden ser utilizados para el desarrollo ágil de software, siempre y cuando el contrato permita el desarrollo simultáneo y la colaboración entre las partes. El primer paso es cambiar el mecanismo de control que favorece el desarrollo secuencial por otro que favorezca al desarrollo concurrente. Después de establecer un diseño conceptual y la capacidad total o global del sistema, se debe esbozar una tentativa de plan de lanzamiento y empezar las iteraciones tan pronto como sea posible para que el cliente pueda ver el código funcionando y ofrecer retroalimentación concreta y oportuna. A medida que se establece la velocidad se debe modificar el plan de lanzamiento y el nivel de recursos si es necesario.
El problema con los contratos de tiempo y materiales es que una vez que el sistema se implementa parcialmente el cliente depende del vendedor, mientras que éste tiene un

incentivo limitado para reducir los costos. El desarrollo ágil atenúa la tendencia de favorecer al vendedor debiendo justificar con valor el dinero gastado al final de cada iteración. En cada iteración, el cliente planifica las características restantes más valiosas, insiste en la entrega del trabajo e integración del código y evalúa el valor entregado. Esto le da al cliente la opción de rescindir el contrato en cualquier momento y aun así obtener el valor de la inversión hasta ese momento.
Si se analiza esto, el desarrollo simultáneo es un enfoque más seguro para los contratos de tiempo y materiales que el desarrollo secuencial y sus controles asociados. El intercambio de valor incremental para el pago gradual protege tanto del vendedor como al cliente. Sin embargo, éste enfoque requiere que los sistemas de gestión de proyectos comúnmente usados para el desarrollo secuencia se dejen de lado. Más importante aún, debe existir la colaboración permanente entre proveedores y clientes.

5.7.3.3.3. CONTRATOS MULTIETAPAS

Los contratos multietapas intentan hacer frente a las incógnitas y riesgos inherentes de los contratos de precio fijo, igualando los riesgos del dinero gastado con el tiempo. Hay dos tipos de contratos: los destinados a conducir un amplio contrato de precio fijo y los que conservan su carácter multietapa a lo largo del proceso.
Los contratos multietapa que se transforman en grandes contratos de precio fijo se inician con uno o dos contratos cortos para conocer lo suficiente sobre el problema y permitir una oferta de precio fijo en el sistema global. Por lo general, sólo un vendedor está involucrado, por lo que este tipo de contrato no es generalmente apropiado cuando se requiere una licitación. Suponiendo que el vendedor es el mismo, el cliente y el vendedor incrementan sus conocimientos, reduciendo el riesgo de grandes sorpresas para ambos. Sin embargo, el incentivo para congelar las especificaciones y no permitir cambios en la etapa final es mayor. Habrá menos simpatía por un cambio en la especificación si se pagó al vendedor para hacer las cosas bien en las primeras etapas. Por lo tanto, este tipo de contrato retiene los problemas de un contrato de precio fijo, si la incertidumbre o los cambios surgen después de que se pactó éste.
El segundo tipo de contrato multietapa, que conserva su carácter de etapas múltiples a lo largo del proceso, presenta una buena oportunidad para los métodos ágiles, ya que es fácil de adaptar al desarrollo iterativo. Sin embargo, estos contratos no están exentos de riesgos; el mayor es que cada parte tiene frecuentes oportunidades para abandonar la relación. Los contratos multietapas crean lo que podría llamarse un monopolio bilateral, es decir que ambas partes llegan a depender la una de la otra.[96] Si una de las partes concluye su participación, la otra parte puede tener mucho que perder.
Una forma de mitigar el riesgo que representa el monopolio bilateral en los contratos multietapas es brindar valor con cada incremento en proporción al dinero gastado. Al igual que en los contratos de tiempo y materiales, es una buena idea implementar primero las características de mayor prioridad para el cliente y entregar código integrado y en funcionamiento con cada iteración.
Otra forma de reducir el riesgo de rescisión de un contrato multietapa es abordar el riesgo a través de la relación, es decir, que las partes construyen la confianza de que la relación continuará siempre y cuando el valor esperado sea entregado. Éstos son contratos de cronograma fijo y alcance variable en el que el cliente evalúa el valor entregado luego de cada iteración. Si el trabajo es aceptable, el contrato continúa con la siguiente iteración.

Aunque no existe una obligación contractual para que ambas partes continúen trabajando juntas, la confianza mutua progresa en la misma proporción en que aumenta la dependencia uno del otro.
Los contratos multietapas serán rápidamente más costosos si debe ser negociado por cada etapa. Por lo tanto, suelen regirse por un contrato principal negociado en la fase inicial, con las órdenes de trabajo ejecutadas para cada iteración.[97]

5.7.3.3.4. CONTRATO DE OBJETIVO DE COSTOS

El problema con los contratos de precio fijo tradicionales es que estimulan el comportamiento egoísta por parte del cliente y el comportamiento defensivo por parte del vendedor. El problema con los contratos de tiempo y materiales tradicionales es exactamente lo contrario: Fomentan el comportamiento egoísta por parte del vendedor y el comportamiento defensivo y orientado al control por parte de los clientes. Lo que se necesita es un punto medio en el que se comparte el riesgo y ambas partes tienen incentivos para tener en cuenta los intereses generales de la iniciativa conjunta.
No hay respuestas enlatadas al dilema del contrato, porque en definitiva, ningún contrato puede prevenir completamente que las partes tomen ventaja una de la otra. Los contratos no generan confianza en que la otra parte cumplirá sus compromisos y no explotará vulnerabilidades. Sin embargo, hay formas de contrato que hacen más fácil para las partes compartir los problemas y beneficios generados por su relación. Un ejemplo de esto es un contrato de objetivos de costos, que si bien no es una panacea, por lo menos es una plataforma en la que una alianza puede ser construida.
Los contratos de objetivos de costos están estructurados de modo tal que el costo total (incluyendo cambios) es una responsabilidad conjunta de los clientes y vendedores. La diferencia principal con un contrato de precio fijo es que si se excede el costo objetivo, ambas partes terminarán pagando más; en cambio si el costo total está por debajo de lo planeado, ambas partes compartirán los beneficios. En cambio, se diferencia de los contratos de tiempo y materiales en que los vendedores no obtienen beneficio adicional si trabajan más tiempo, pero es posible que reciban un beneficio si están por debajo del costo o el cronograma.
En un contrato de desarrollo de software con objetivos de costos, las partes comienzan con un acuerdo general de lo que está por llevarse a cabo, reconociendo que los detalles no se conocerán hasta que el trabajo mutuo esté realizado. Luego llegan a un acuerdo sobre el costo objetivo para el sistema y pactan un cronograma. En este tipo de contrato se entiende que el costo objetivo es muy importante, por lo que el diseño y las características detalladas se enfocarán en satisfacer el costo objetivo. Existe un compromiso de ambas partes para cumplir el costo objetivo y se entiende que esto requerirá un esfuerzo conjunto tanto de técnicos y usuarios de ambas partes.
Un contrato de objetivos de costo reconoce que los costos reales no son necesariamente los mismos que los planeados, por lo que prevé un reparto equitativo de cualquier costo que supere el objetivo o una distribución justa de los beneficios si los costos son inferiores a los planeados. Estos contratos deben dar al cliente un incentivo para mantener la demanda de características acordes a los costos objetivos, al mismo tiempo que incentivan a los vendedores de completar el trabajo por debajo del costo. Por lo general, el incentivo del

cliente es proporcionado mediante una cláusula que asegura la negociación del reparto equitativo de los costos en el caso de que varíen significativamente de los objetivos. Uno de los siguientes puntos por lo general establece el incentivo del vendedor:

Pago fijo por costo adicional: El costo objetivo no incluye beneficio para el vendedor; se incluye una tarifa por separado para proporcionar beneficio al vendedor. La tasa se paga por lo general después de que el trabajo se ha completado con éxito. Si el costo total excede el costo objetivo, el vendedor trabaja al costo por el resto del contrato. Si el costo total es inferior al pautado, el vendedor recibe un mayor margen de ganancia. Se debe incluir un bono por mantenerse debajo del costo objetivo.

Beneficio por no excederse: El costo objetivo incluye la ganancia del vendedor. Se compromete a reducir los precios y excluir su ganancia luego de haber alcanzado el costo objetivo. Si el costo total es excesivo, el vendedor trabaja al costo. Sin embargo, en este caso, el vendedor no tiene ningún incentivo para mantenerse por debajo del costo que se persigue a menos que haya una bonificación por terminación anticipada.

La parte más valiosa de los contratos de objetivos de costo es que comunican con mayor precisión la intención de la administración a los trabajadores de primera línea de ambas partes y los animan a trabajar juntos para lograr el propósito. Si las expectativas de costo no se ponen en conocimiento de los equipos de trabajo desde el principio, es muy poco probable que el diseño resultante encuentre el objetivo. Los contratos de objetivos de costo deben dejar los detalles del alcance a la discreción de los equipos técnicos, ya que la reducción del alcance es terreno más fértil para el control de costos.

5.7.3.3.5. CONTRATO CON CRONOGRAMA DE OBJETIVOS

Algunas veces el cronograma es más importante que el costo, aunque el costo raramente carece de importancia. Si el número de personas que trabajan en el sistema no varía y no se compra o adquiere la licencia de ningún componente, entonces el costo objetivo y el cronograma objetivo son la misma cosa.[98] Las compañías de productos de software a menudo cumplen con cronogramas difíciles mediante la fijación de recursos, planificación y trabajando primero en los puntos de mayor prioridad. Cuando el tiempo se acaba, las características de baja prioridad se dejan de realizar, pero el lanzamiento satisface la intención general de marketing del producto.
De la misma forma, un contrato de objetivo de costo por lo general se puede ejecutar como un contrato con cronograma objetivo fijando los recursos y calendario. Las características deberían abordarse por orden de prioridad y en cada iteración se debe entregar software apto para ser implementado, integrado, probado y en funcionamiento. Mucho antes de que se cumpla el plazo, el software debe ser realmente implementado. Entonces, las iteraciones pueden continuar para hacer frente a los problemas que surjan en la producción. Con este enfoque, por definición el trabajo realizado respetará el cronograma, el presupuesto y las características entregadas deberían cumplir con la intención global del contrato.
Si el cronograma es realmente lo único importante, entonces un contrato con cronograma objetivo es más apropiado que un contrato de objetivo de costo. Esto permite al equipo agregar recursos o componentes con licencia para cumplir con el cronograma según sea

necesario. Cuantos más grados de libertad se les da a los trabajadores en el objetivo del contrato, más fácil será para ellos encontrar la manera de cumplirlo.

5.7.3.3.6. CONTRATO DE BENEFICIO COMPARTIDO

Los contratos de objetivo de costos y de cronogramas establecieron un entorno en el que los equipos trabajan eficazmente a través de los límites de la empresa, ya que es claro que ambas compañías compartirán los riesgos y beneficios del trabajo. Esta es la clave de los contratos de colaboración; las personas que realizan el trabajo deben percibir que ambas partes tienen interés en los resultados de sus esfuerzos. Un contrato de beneficio compartido es otro mecanismo eficaz para compartir riesgos y ganancias si se está desarrollando productos para la venta.
En un contrato de origen conjunto ambas compañías comparten la responsabilidad de desarrollar un sistema y también se espera que el vendedor transfiera su experiencia al cliente. Este contrato es exitoso si el vendedor trabaja ayudando a los clientes a desarrollar la capacidad de hacer el trabajo ellos mismos. El origen conjunto es un enfoque fundamentalmente de colaboración, por lo que este tipo de contratos no tienden a crear motivación para el comportamiento de auto servicio.

5.7.3.4. ALCANCE OPCIONAL

Se han observado varios tipos de contratos que se pueden trabajar para el desarrollo ágil de software:

- Contratos de tiempo y materiales que utilizan el desarrollo concurrente implementando primero las características de más alta prioridad y entregando código integrado y en funcionamiento en cada iteración, de manera tal que el cliente puede administrar fácilmente el costo limitando el ámbito.
- Contratos multietapa que utilizan un contrato principal y órdenes de trabajo para entregar cada iteración, con similar énfasis en el desarrollo concurrente implementando características de más alta prioridad y entregando el código integrado en cada iteración.
- Contratos de objetivo de costo que reúnen a los trabajadores de primera línea de ambas partes para trabajar juntos en busca de la solución al problema que representa un costo objetivo, dándoles la libertad de limitar el ámbito como mecanismo fundamental para lograrlo.
- Contratos de beneficio compartido que asumen que con el tiempo, las partes modificarán los que están haciendo para alcanzar el beneficio mutuo.

Aquí existe un tema común: Todos estos contratos son mecanismos que evitan la fijación del ámbito en detalle. Esto no debería ser una sorpresa, ya que normalmente más del 50 por ciento de las prestaciones de un sistema típico se utilizan rara vez o nunca, lo que sugiere que el terreno más fértil para la mejora de la productividad en el desarrollo de software radica en no implementar características que no son necesarias.[99] La mejor manera de desarrollar software de bajo costo y alta calidad es escribiendo menos código.[100] Contratar un equipo de desarrollo de software para lograr un propósito dentro de las limitaciones de

costos y cronogramas es casi lo mismo que pedirles que descubran qué características dejar fuera del sistema.
La sabiduría popular sostiene que especificar y controlar el alcance de un contrato es necesario para proteger a una organización del comportamiento egoísta de la otra parte. Sin embargo, el efecto de esta protección es una cadena de valor sub optimizada. Aunque parezca contradictorio, un control rígido del alcance tiende a expandir y no reducir el ámbito. Esto a su vez conduce a un aumento significativo en el costo de las características, así como el costo del sistema de control. Las organizaciones que utilizan la subcontratación como una forma de ahorrar dinero en general podrán lograrlo si colaboran con los vendedores mediante el uso de algún tipo de contrato de alcance opcional.
El establecimiento de una relación de sociedad con los proveedores por lo general ocurre en las primeras etapas del proyecto y no es tan simple como usar cualquier forma específica de contrato. Ambas partes necesitan una clara comprensión del valor que podrían aportar a la otra parte si se enfocan en el beneficio mutuo en lugar del beneficio individual. Las asociaciones requieren prácticas coherentes de manera que los socios desarrollen la confianza de que los compromisos serán honrados y las vulnerabilidades no serán explotadas, incluso si las personas involucradas cambian. Esto a su vez requiere de acuerdos creativos que no tratan de cubrir toda eventualidad, pero en cambio ofrecen maneras de lidiar con los eventos futuros impredecibles de manera que ambas partes lo perciban como justos y equitativos.

5.7.3.5. METODOLOGIA DE APLICACIÓN

A lo largo del desarrollo del presente trabajo se han consultado y analizado numerosas fuentes que abordaron los diferentes conceptos involucrados en el desarrollo de software Lean y analizándolos desde variados puntos de vista. Pero todas en algún punto convergen en la misma interrogante:
¿Lean y Ágil son dos nombres para la misma cosa?
Esta ha resultado ser una pregunta común que se ha transformado en un concepto erróneo. La respuesta es simplemente **NO**.
Es fácil ver porque existe esta confusión. Lean y Ágil comparten los mismos objetivos: aumentar la productividad del desarrollo de software al mismo tiempo que se incrementa la calidad del producto resultante. Para aumentar la confusión, las prácticas que componen las diversas metodologías agiles también son compatibles con los principios Lean.
El pensamiento Ágil tiene una perspectiva diferente de la metodología Lean, generalmente con un enfoque más limitado. Por el contrario, Lean tiene una visión más amplia, mirando todo el contexto empresarial en el que se lleva a cabo el desarrollo de software. El pensamiento Lean toma a las metodologías Agiles de desarrollo de software como prácticas de apoyo válidas del desarrollo de software Lean.
En el desarrollo del marco teórico hemos ahondado en los siete principios que sustentan el pensamiento Lean:

- Eliminar desperdicios
- Construir la calidad
- Crear conocimiento
- Posponer los compromisos
- Entregar rápido

- Respetar a las personas
- Optimizar el todo

A continuación analizaremos cada uno de estos principios desde una óptica específicamente referida a su aplicación práctica, lo que determinará la metodología planteada.

5.7.3.6. ELIMINAR DESPERDICIO

Es el principio más amplio y el más importante de la metodología. Veremos cuáles son los equivalentes de defectos, sobreproducción, transporte, esperas, inventario, movimiento y procesamiento para el desarrollo de software.

Defectos → Defectos: un defecto es un defecto, ya sea que se esté hablando de la fabricación de un producto físico o desarrollando software. Generan la necesidad de volver a realizar el trabajo, lo cual es costoso y se considera desperdicio que no genera valor agregado. El objetivo en un entorno Lean es la prevención de los defectos, mientras que el desarrollo tradicional se centra en la búsqueda de los defectos después de que ya se han producido, siendo especialmente caros cuando se detectan tarde.
En la metodología Lean, cuando se encuentra un defecto, la respuesta es encontrar su causa y hacer los cambios que aseguren que no pueda repetirse. En el desarrollo de software esto significa tener un conjunto de pruebas automatizadas que prevengan los defectos que puedan generarse en el producto sin ser detectados. Cuando un defecto aparece, se crea una nueva prueba para detectarlo, de forma tal que no podrá pasar sin ser descubierto de nuevo.

Sobreproducción → Características Adicionales: cada línea de código cuesta dinero. Durante la vida útil del software, el costo de escribir el software probablemente es el menos significativo. El código también debe ser diseñado, documentado y mantenido (modificado, mejorado, depurado, etc.). Esto significa que un grupo de miembros actuales y futuros del equipo deberán leer y comprender el código varias veces. Su presencia debe ser tenida en cuenta en cada futuro cambio del producto o mejora.
La regla 80/20 se aplica en la mayoría de los productos de software: el 80% de las necesidades reales del usuario se proporcionan en un 20% de las características del producto. Esto significa que el otro 80% de las funciones de un producto son rara vez utilizadas (o nunca). En el año 2002, en una conferencia brindada por el StandishGroup, se reportó que el 45% de las características nunca fueron utilizadas y que solo el 20% se usaron a menudo o siempre. De hecho, otro documento presentado en el 26º Workshop de Ingeniería de Software de la IEEE informó que sobre 400 proyectos de desarrollo examinados en un periodo de 15 años, menos del 5% del código era realmente útil o usado.
Peor aún, cuando el StandishGroup se enfocó solamente en proyectos exitosos en su estudio titulado CAOS, encontraron que el 64% de las características fueron raramente o nunca utilizadas (Figura 11-1). Esto es un enorme desperdicio que se convierte en una creciente fuga de los recursos del proyecto. El tiempo dedicado a desarrollar estas características sería mucho mejor utilizado si se emplea para trabajar en las necesidades reales del cliente. Si una característica no aborda una clara necesidad del cliente, no debe ser creada.

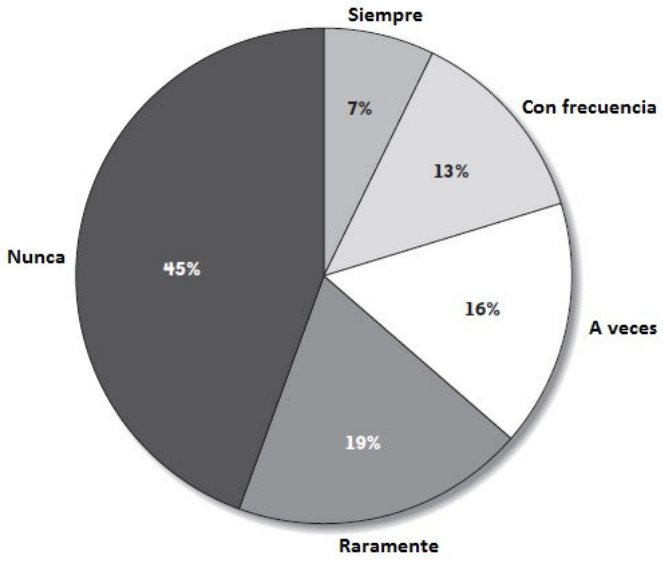


Figura 5.14 Porcentaje de características entregadas realmente usadas

Transporte → Transferencias: cuando se trabaja en grandes proyectos, las siguientes situaciones pueden ser familiares:

- Los analistas crean un documento que contiene todos los requisitos del producto y es entregado a los arquitectos.
- Los arquitectos toman los requisitos y crean el diseño del producto, el cual se pasa a los programadores.
- Los programadores escriben el código para implementar el diseño y lo entregan a los probadores de control de calidad.
- Los probadores validan el producto resultante en función de los requisitos.

Este es un clásico proceso en cascada repleto de transferencias. A través de cada una de ellas, una enorme cantidad de conocimiento se pierde simplemente porque no es posible grabar todo lo que se ha aprendido, descubierto, creado y conocido en una forma escrita. Una gran cantidad de conocimiento tácito no se transmite.
Esto significa que los arquitectos no van a entender los requisitos tan profundamente como los analistas y que los programadores no van a entender el diseño como los arquitectos. Esta comprensión incompleta dará lugar a errores y omisiones que requerirán costoso trabajo para su corrección. Peor aún, la pérdida de conocimiento se agrava con cada traspaso.

Esperas → Retrasos: en los proyectos de desarrollo de software casi constantemente se toman decisiones. Si un desarrollador tiene un amplio conocimiento del sistema que se creó, él ya sabrá las respuestas de la mayoría de sus preguntas (o será capaz de deducirlas). Sin embargo, el desarrollador no puede saber todo y siempre necesitará hacer preguntas a sus compañeros de trabajo, clientes y otras partes interesadas. Si estas personas están disponibles de inmediato, no hay ningún retraso y el desarrollo continúa a toda velocidad.

Cuando una pregunta no se puede responder de inmediato, el escenario está preparado para todo tipo de desperdicios como resultado. El desarrollador podría:

- Suspender la tarea actual y pasar a otra cosa, lo cual se conoce como conmutación de tareas.
- Adivinar la respuesta, lo que genera trabajo adicional cuando la suposición es incorrecta. La cantidad de trabajo extra aumenta a medida que el error se descubre más tarde.
- Tratar de encontrar la respuesta; pero incluso cuando se intenta encontrarla, si es muy difícil se terminará adivinando la respuesta para evitar complicaciones.

No importa cual escenario se elija, en todos hay desperdicio.

Inventario → Trabajo Parcialmente Hecho: en pocas palabras, nos referimos a algo que se ha iniciado pero no terminado. Podrían ser requisitos (características) que no han sido codificados, código que no se ha probado, documentado ni desplegado o errores que no han sido corregidos. En lugar de dejar que el trabajo parcialmente hecho se acumule en las colas, el enfoque Lean utiliza un único flujo para implementar una característica lo más rápido posible.
Una característica no está completa hasta que sea potencialmente desplegable, totalmente documentada, probada y libre de errores.

Movimiento → Conmutación de Tareas: la conmutación e interrupción de tareas matan la productividad. Se necesita tiempo para enfocar la mente en la tarea, poder entender los factores necesarios y comenzar con el proceso de resolución de problemas. Las interrupciones reinician este proceso y la conmutación de tareas hace que se deba "volver a aprender" antes de siquiera comenzar a ser productivo de nuevo.
Esta es la razón por la que un único flujo es tan productivo. Se puede trabajar por completo a través de una característica o una tarea sin el desperdicio de la conmutación.

Procesamiento Adicional → Procesos Innecesarios: podemos calificarlos como desperdicio puro. Se interponen en el camino de la productividad sin añadir ningún valor. Incluyen los procedimientos que no logran nada y documentación que nadie lee. También podemos considerar las tareas manuales que pueden ser automatizadas y procedimientos que dificultan tareas sencillas.

5.7.3.7. CONSTRUIR LA CALIDAD

Uno de los conceptos clave introducidos por TaiichiOhno en la manufactura fue que no se puede inspeccionar la calidad de un producto al final de la línea de producción. Ese enfoque detecta problemas pero no hace nada para corregirlos. En cambio, cada paso en el proceso debe ser a prueba de errores y auto inspeccionarse. Cuando se encuentra un problema, toda la línea de montaje se detiene hasta que la raíz del problema es encontrada y corregida (por lo tanto no puede ocurrir de nuevo).
Un ejemplo famoso es la fábrica de automóviles New United Motor Manufacturing Inc. (NUMMI), una empresa conjunta entre Toyota y General Motors. Se les dijo a los trabajadores que hicieran un buen trabajo y que detuvieran la línea cada vez que algo les

impidiera hacerlo. Tomó casi un mes producir el primer vehículo, pero desde que cada problema se resolvió una sola vez (permanentemente) la planta se convirtió rápidamente en líder en calidad y productividad en Estados Unidos.
El desarrollo de software tradicional ha seguido el mismo patrón que la fabricación de automóviles tradicional americana: permitir a los defectos deslizarse y atraparlos más tarde por medio de las inspecciones de control de calidad. El enfoque Lean se basa en hacer al código a prueba de errores mediante la escritura de pruebas a medida que se codifican las características. Estas pruebas evitan los cambios posteriores en el código al introducirse defectos no detectados.

5.7.3.8. CREAR CONOCIMIENTO

El punto aquí es no olvidar las lecciones que se han aprendido. Obviamente, cometer los mismos errores una y otra vez o volver a aprender cómo funciona algo es un desperdicio de tiempo y esfuerzo.
Se debe encontrar la manera de registrar el conocimiento del equipo para que pueda localizarse fácilmente la próxima vez que se lo necesite. Es difícil ser especifico respecto a esto porque lo que tiene sentido y lo que dará resultado depende en gran medida del contexto. Sin embargo, generalmente es mejor almacenar una determinada pieza de conocimiento lo más cerca posible de su fuente.
Por ejemplo, si se supone que se está agregando una característica a un sistema y se debe leer el código para comprender el funcionamiento de un subsistema. Lo que se aprende debe ser registrado en alguna parte. Se podría agregar la nueva información a un documento de diseño detallado, pero sería mucho más útil registrarlo como un comentario en el código. Después de todo, la próxima vez que alguien necesite conocer esta información, es probable que también busque en el código.
De la misma manera, en la arquitectura, el diseño y el código siempre se deberán considerar alternativas y tomar decisiones. Cuando se tome una decisión, de debe considerar la posibilidad de registrar por qué se ha elegido una alternativa frente a otra. A veces este conocimiento puede ser un ahorro de tiempo útil en el futuro, pero en ocasiones también puede ser excesivo. Siempre se debe usar el mejor criterio y tratar de mantener un equilibrio útil.

5.7.3.9. POSPONER LOS COMPROMISOS

Las mejores decisiones se toman cuando se tiene la mayor cantidad de información disponible. Si no se debe tomar una decisión de manera urgente, lo mejor es esperar hasta tener más conocimiento e información. Pero no se debe esperar demasiado tiempo, de modo que la falta de decisión retrase otros aspectos del proyecto.
Se debe esperar hasta el último momento responsable para tomar una decisión irreversible.
Tomemos el caso en donde se debe tomar una decisión arquitectónica. En primer lugar, debería determinarse cuando es el último momento responsable para esa decisión. Utilizar el tiempo intermedio para acumular conocimiento sobre las necesidades reales de los otros componentes del sistema. Se debe usar ese tiempo para explorar las características de las opciones alternativas. Por lo tanto, el mejor enfoque es tratar todas las posibilidades en caso de poder hacerlo y, finalmente, elegir la que mejor se adapte a las necesidades del sistema.

Esto se conoce como *Desarrollo Basado en Conjuntos* (tema tratado en detalle en secciones anteriores). Con un diseño basado en conjuntos se persigue simultáneamente múltiples soluciones para finalmente elegir la mejor.
Un ejemplo clásico de este enfoque fue el diseño del Toyota Prius. Los requisitos para el automóvil no especificaban un motor hibrido; solo indicaron que debía tener un rendimiento de combustible excepcionalmente bueno. Compitieron varios diseños de motores que se desarrollaron simultáneamente. El hibrido fue elegido en el último momento responsable (cuando solo quedaba el tiempo suficiente para ingresar a producción al Prius y respetar la fecha de lanzamiento establecida).
Esto podría parecer desperdicio, pero en realidad, tomar la decisión equivocada podría haber reducido drásticamente el éxito del Prius, lo que hubiera generado el desperdicio que implica una oportunidad perdida.

5.7.3.10. ENTREGAR RAPIDO

El desarrollo de software es una tarea abstracta. Sin embargo, la mayoría de las personas (incluidos los clientes) trabajan mejor cuando tratan con cosas concretas. Cuando podemos ver, tocar y sentir algo, se convierte en real y nuestro cerebro puede pensar más fácilmente sobre lo que funciona y lo que no. Nuestra imaginación puede soñar características o capacidades que no nos dimos cuenta que necesitábamos.
Por esta razón, los requisitos del software son tan volátiles. En enfoque en cascada obligaría a esperar hasta el final del proyecto para obtener retroalimentación de los clientes en función del uso real del software y es por eso que este proceso es tan propenso al fracaso.
“Entregar rápido” significa desarrollar características en pequeños lotes que se entregan a los clientes de forma rápida, en iteraciones cortas. Estas características pueden ser implementadas y entregadas antes que los requisitos asociados puedan cambiar. Esto significa que el cliente tiene una oportunidad de usar esas funciones para brindar retroalimentación que puede ser utilizada para cambiar los otros requisitos antes de su implementación.
Completar cada iteración corta proporciona una oportunidad de cambiar los requisitos y su prioridad en función de la retroalimentación y su uso. El resultado final es un producto que satisface mejor las necesidades reales de los clientes, eliminando al mismo tiempo la enorme cantidad de desperdicio y trabajo adicional creado por los cambios en los requisitos.

5.7.3.11. RESPETAR A LAS PERSONAS

Respetar a las personas significa confiar en que sabrán la mejor manera de hacer su trabajo, permitiéndoles participar para exponer las fallas del proceso actual y animándolos a encontrar maneras de mejorar sus trabajos y los procesos relacionados. El respeto por las personas significa darles reconocimientos por sus logros y solicitar activamente sus consejos. De esta manera se cuida el recurso más valioso del equipo: la mente de sus integrantes.

5.7.3.12. OPTIMIZAR EL TODO

Esta es una parte importante del pensamiento Lean sin importar donde se lo esté aplicando. Siempre que se optimiza un proceso local, casi siempre se lo está haciendo a expensas de toda la cadena de valor (esto es sub optimización).
Si no se tiene control sobre toda la cadena de valor, es posible que de manera forzosa deba sub optimizarse una parte de ella. Sin embargo, generalmente se debe tratar de incluir la mayor parte posible de la cadena de valor cuando se trata de optimizar el proceso.

Después de haber recapitulado los conceptos que sustentan los principios del desarrollo de software Lean y haciendo referencia a la pregunta hecha al principio de la sección, inevitablemente surge la pregunta:

¿Cuál es la diferencia entre el desarrollo de software Lean y el Ágil?

Como respuesta podríamos decir que en la superficie, no muchas cosas.
Tanto Lean como Ágil tienen por objeto mejorar la calidad del software (de la manera en que es percibida por el cliente) así como la productividad del proceso de desarrollo. Ambas valoran los cambios en los requisitos que prácticamente con certeza ocurrirán en el transcurso del proyecto y dan la mayor importancia a la entrega de software que cumpla con las necesidades reales del cliente (no las necesidades iniciales percibidas por el cliente).
La diferencia está en la perspectiva subyacente y la filosofía. Dicho en otras palabras: la manera de pensar.
La filosofía Ágil en su mayoría se ocupa de la práctica específica del desarrollo de software y la gestión de proyectos que rodea el proceso. Los métodos agiles por lo general no se preocupan por el contexto de negocio circundante en donde el desarrollo se está llevando a cabo.
Por el contrario, los principios Lean pueden aplicarse en cualquier ámbito, desde la práctica específica del desarrollo de software hasta toda la empresa donde en donde el desarrollo de software es solo una pequeña parte. Incluso es común en la manufactura Lean ir hacia afuera de la compañía e incluir a los proveedores en el alcance de la mejora de procesos Lean. Cuanto mayor sea el alcance, mayores serán los beneficios potenciales.
La mayoría de los esfuerzos Lean empiezan desde abajo y amplían su alcance a través del tiempo, descubriendo más y más beneficios en el proceso. En cualquier caso, se puede decir con seguridad que Lean considera que todas las metodologías Agiles son prácticas de apoyo válidas.
El enfoque principal del desarrollo Ágil apunta a la estrecha colaboración con el cliente y la rápida entrega de software funcional lo más pronto posible. Lean considera que eso es importante, pero su atención se centra principalmente en la eliminación de los residuos en el contexto de lo que valoran los clientes.
Una herramienta clave que posee Lean para la detección y eliminación de residuos es el mapa de flujo de valor (MFV). Se trata de un diagrama en forma de mapa de todas las actividades que tienen lugar de principio a fin, por ejemplo, desde que el cliente solicita una nueva función hasta que la misma se entrega al cliente. Cada paso entonces se identifica como *valor agregado* (desde la perspectiva del cliente), *sin valor agregado* o *sin valor agregado pero necesario.*

Por último, Ágil tiene un gran número de metodologías formales, mientras que Lean no las posee. En cambio, Lean tiene un kit de herramientas conformado por prácticas recomendadas para elegir.
De hecho, al implementar el desarrollo de software Lean, es bastante común elegir una metodología Ágil liviana como punto de partida y comenzar a aplicar otras herramientas Lean (como MFV) a partir de allí.

5.8. PRÁCTICAS APLICABLES

El desarrollo de software Lean ofrece poca orientación más allá de brindar un cúmulo de posibles prácticas. Esto significa que el desarrollador debe decidir qué usar (o no usar) y cuando hacerlo, lo cual adquiere cierto grado de dificultad cuando no se posee el conocimiento y la experiencia para tomar esas decisiones. Sin embargo, a continuación se presentarán una serie de prácticas que son universalmente encontradas en casi la totalidad de las metodologías Agiles e implementaciones de desarrollo de software Lean. Cada práctica por si misma brinda beneficios de productividad y/o calidad y pueden adoptarse en cualquier orden que se desee (con un par de excepciones). Por lo tanto, el orden en que se presentan a continuación no es necesariamente obligatorio.

5.8.1. ADMINISTRACION DEL CODIGO FUENTE Y COMPILACIONES CON SCRIPTS

En nuestro contexto, el término “práctica” será utilizado para referirse a algunos procedimientos que podrían ser utilizados si se está tratando de adoptar las prácticas Lean. Podrían ser considerados requisitos previos necesarios, pese a no ser prácticas Lean propiamente dichas y por eso nos referimos a ellos como prácticas de nivel cero. Se trata de prácticas fundamentales que se necesitan saber antes de intentar cualquier mejora adicional.
Las dos prácticas de nivel cero que estamos hablando aquí son la **Administración del Código Fuente** (SCM por su sigla en inglés)y**Compilaciones con Scripts**.

5.8.1.1. ADMINISTRACION DEL CODIGO FUENTE

También conocido como control de revisión o control de versiones, básicamente significa mantener todo el código fuente y otros artefactos del proyecto en un repositorio central que mantiene una versión de la historia completa de cada archivo.
Hay dos tipos de sistemas de control de versiones: los centralizados y los distribuidos. Se abordarán los conceptos básicos utilizando un sistema centralizado, ya que estos son utilizados más ampliamente.
Aunque cada sistema SCM tiene su propia nomenclatura, especialmente para los conjuntos más avanzados de comandos, hay varias operaciones básicas comunes a todos los sistemas. Cada vez que se quiera obtener el código actual de un proyecto existente, se puede revisar una versión de ese repositorio. Se puede añadir, modificar y eliminar archivos mediante la comprobación de los cambios. También se puede actualizar mediante la descarga de los cambios que se hayan hecho en el repositorio desde su revisión o última actualización.

5.8.1.2. BENEFICIOS

Poder tener versiones es la capacidad más básica e importante que un sistema SCMofrece. Dicho sistema almacena un historial de revisiones completo de todos los cambios realizados en cada archivo. A cada modificación se le asigna un número de revisión y se puede acceder a las versiones anteriores en una gran cantidad de formas convenientes. Si se registra un cambio y luego se descubren algunas consecuencias no deseadas, será fácil volver a la versión anterior del trabajo.
Si se descubre un error, se puede ir temporalmente a una versión anterior a su introducción y luego hacer un *diff* (utilidad para la comparación de archivos que genera las diferencias o cambios entre ellos) de ambas versiones para ver exactamente lo que ha cambiado. Es relativamente sencillo realizar un *diff*entre versiones actuales y anteriores de un solo archivo o incluso todo el repositorio.
En la mayoría de los sistemas de administración de código, se pueden utilizar etiquetas para marcar una versión específica del código. Esto es realmente útil si se quiere marcar un lanzamiento del software. Se puede volver a una versión etiquetada del mismo modo en que se puede volver a una fecha o un número de revisión. La posibilidad de acceder fácilmente a las versiones anteriores no es sólo una conveniencia; proporciona una red de seguridad que permite al equipo de trabajo realizar cambios en el software sin temores.
Un sistema SCM también hace que sea más fácil colaborar y comunicarse con eficiencia en un ambiente de equipo. Cada vez que un desarrollador registra un cambio, el sistema almacena metadatos acerca de quién hizo la revisión y cuando ocurrió. Estos datos pueden ser buscados y ayudar en una serie de situaciones. Por ejemplo, si un nuevo miembro del equipo está comenzando a trabajar en una sección de código, puede mirar a través de la historia para saber quien creó el archivo y quien ha ido modificándolo con el tiempo. Se puede utilizar estos datos para determinar la persona indicada para solicitar una revisión por pareso con la cual formar equipo para enfrentar una decisión de diseño o un error difícil.
La administración del código ofrece un espacio de trabajo compartido que ayuda al equipo a documentar y compartir diseños e ideas. La documentación es más útil cuando se mantiene al día y es muy cercana al código. Si el equipo almacena documentación y artefactos de diseño junto con el código en el repositorio, es más probable que sean usados y se mantengan actualizados, a la vez que tendrán también el mismo historial de revisión que el código. Esto significa que si se extrae una copia del código base para una fecha o etiqueta en particular, se obtendrán las versiones adecuadas del código y la documentación correspondiente.

5.8.1.3. ADMINISTRACIÓN DE CÓDIGO FUENTE CENTRALIZADA

La mayoría de los sistemas convencionales de administración de código fuente como el *Sistema de Versiones Concurrentes (SVC)*, *Subversión* y *ClearCase* son centralizados, es decir, que utilizan un único repositorio centralizado para almacenar sus archivos. Los equipos suelen alojar este repositorio en un servidor accesible para todos, proporcionando permisos completos de manera que puedan registrar (para contribuir) y actualizar (para sacar el último código que el resto del equipo ha incluido en el repositorio). La figura 12-1 muestra este tipo de sistema.

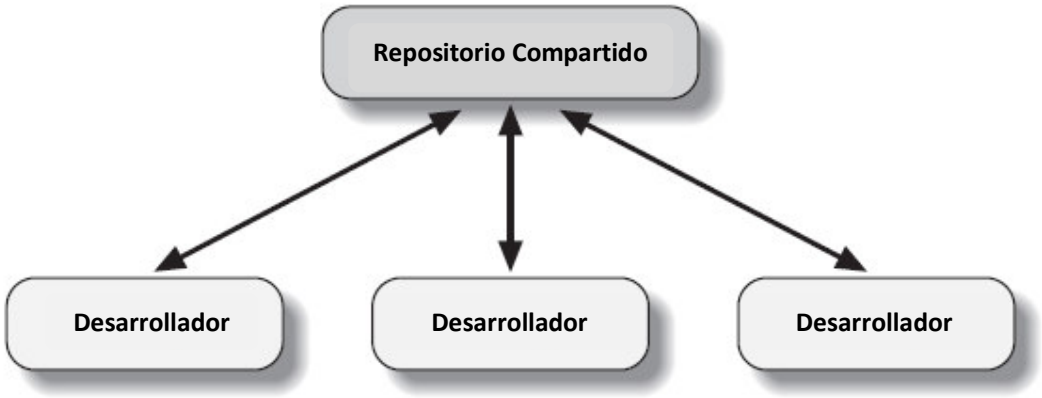


FIGURA 5.15 Archivo centralizado

Los sistemas de administración de código fuentecentralizados han existido desde la década de 1970. Dado que son sistemas centrales en los que todo el equipo se apoya, estos tienden a mantenerse por un largo tiempo. De los principales sistemas que todavía se emplean, los SVC son los más antiguos, habiendo ganado popularidad a finales de la década de 1980.ClearCase se introdujo en 1990 para satisfacer las necesidades de las empresas más grandes, mientras que Subversión es el más moderno, introducido en 2000 para mejorar el SVC. Hoy en día Subversión es la opción más popular para los nuevos proyectos.
En los sistemas SCM centralizados, cuando se da salida auna copia del repositorio, se descarga una copia de trabajo local de cada archivo en el depósito. Cuando se hace un cambio, el sistema puede decir que se ha modificado su copia de trabajo, pero la copia en el repositorio central permanece intacta. Esto significa que se puede modificar tantas veces como se desee dicha copia de trabajo sin afectar a ninguna otra persona en su equipo. Cuando esté listo para compartir sus cambios con el equipo, los mismos deben ser subidosy modificar los archivos en el depósito centralizado. Finalmente, cuando el equipo actualiza (descarga la versión más reciente de los archivos del repositorio), obtendrán los cambios desde su entrada.
Como en cualquier ambiente de equipo, es importante comunicarse de manera efectiva para evitar estorbarse mutuamente. Cuando se utiliza un sistema centralizado de administración de código fuente, hay algunas expresiones comunes que la mayoría de los equipos adoptan para mantener todo funcionando sin problemas:

Actualizar antes de registrar cambios: antes de registrarse, se debe probar el cambio frente al código base actual, por lo que siempre se debe actualizar antes de concretar modificaciones. Esto asegurará que no está probando contra código obsoleto. Si el equipo está subiendo modificaciones con frecuencia, es normal que los diversos cambios se introduzcan en el repositorio al mismo tiempo. El sistema SCM alertará de que la copia es antigua si se intenta actualizar un archivo que alguien ha modificado posteriormente a nuestra última actualización. Dicho esto, el sistema sólo se basa en archivos individuales y no en las dependencias de su código.

No romper lo construido: desde que todos en el equipo actualizan desde el mismo código base, los desarrolladores suelen ser animados esperar hasta que sus funciones hayan sido

probadas y puestas en funcionamiento antes de registrarlas.Si se da ingreso al código erróneo, nadie en el equipo de trabajo podrá compilar y ejecutar el software hasta que se solucione el problema. Por lo que la calidad del código y las pruebas afectan directamente la productividad del equipo.

Registrar cambios con frecuencia: Si se quiere seguir con la idea de no romper lo construido, se debe descomponer la implementación en pequeños cambios para evitar trabajar durante largos períodos de tiempo sin la red de seguridad del control de versiones. Se debe realizar la actualización después de cada uno de esos pequeños cambios. Si se llegara a romper la compilación, el pequeño alcance del cambio hará que sea más fácil localizar y corregir la fuente del problema.

Si se puede, el sistema fusionará automáticamente para cada vez que se registre un cambio. Normalmente, el sistema puede fusionar un archivo de forma automática, siempre y cuando no haya dos cambios en el mismo lugar. Si eso sucede, el sistema cede la responsabilidad al desarrollador.

5.8.1.4. SCM DISTRIBUIDO

Los sistemas de administración de código fuente distribuidos, como Git y Mercurial, son mucho más nuevos que sus contrapartes centralizadas. Fueron creados para satisfacer las necesidades de grandes proyectos de código abierto, que tienen grandes bases de código y equipos distribuidos de desarrolladores. Las soluciones innovadoras que se implementan apuntan hacia algunos de los problemas de los equipos de todos los tamaños corren cuando se utiliza en SCM centralizada. Dicho esto, su relativa inmadurez hace que sea difícil recomendar su uso para equipos nuevos en el concepto de administración de código fuente.
La mayor diferencia entre los sistemas centralizados y distribuidos es que en el segundo no se verifica o descarga una copia de trabajo de los archivos; en cambio, se clona o descarga una copia de todo el repositorio en funcionamiento. Esto incluye todo el historial de revisiones, lo que da la capacidad de realizar prácticamente todas las operaciones de SCM a nivel local, sin necesidad de conectarse a un servidor central. En otras palabras, un sistema distribuido permite trabajar en la copia local del repositorio.

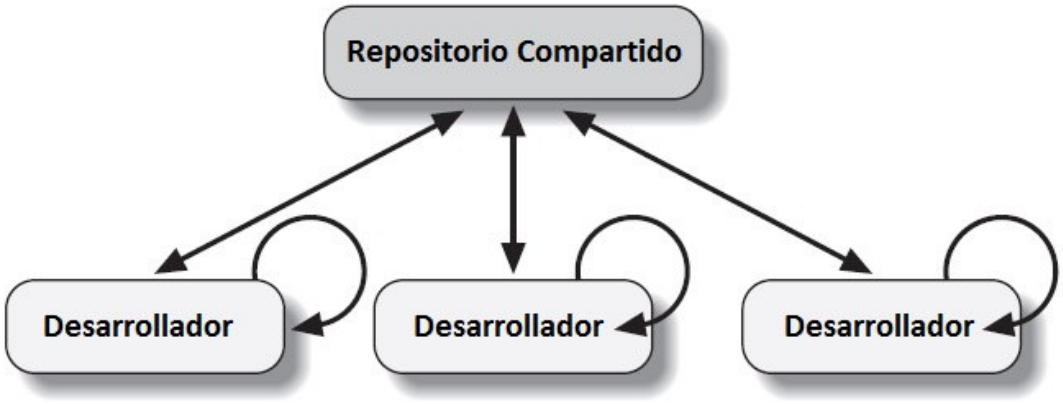


FIGURA 5.16 Un flujo de trabajo centralizado en un sistema distribuido SMC

En la práctica, los cambios no se han aplicado a menudo y se controlan de una línea a la vez. Los desarrolladores a menudo terminan haciendo modificaciones en varios lugares dentro de un archivo o en varios archivos para implementar un solo cambio. En un sistema centralizado, todos estos cambios van totalmente dentro de una subversión porque el desarrollador no los registra hasta que se comprueba el cambio completo en el repositorio central. En un sistema distribuido, ya que se tiene una copia de trabajo del repositorio completo, se puede probar en la copia local tan a menudo como desee sin afectar a nadie más en el equipo. Esto permite sacar el máximo provecho de la red de seguridad SCM, incluso si se queda atascado haciendo un gran cambio antes de registrarse en el repositorio central. Cuando se haya completado el cambio, se lo envía desde el repositorio local hasta el repositorio central para compartirlo con el resto equipo.
Un sistema SCM distribuido también tiene otras ventajas y desventajas que sólo mencionaremos brevemente. La velocidad de las operaciones comunes de SCM es extremadamente rápida debido a que operan en su copia local del repositorio y no requieren ningún acceso a la red. Los sistemas distribuidos también pueden escalar a equipos más grandes y ad hoc mediante el apoyo a los flujos de trabajo que no se basan en un único repositorio central compartido. La desventaja para cada desarrollador en un sistema distribuido es que tiene su propia copia de trabajo de todo el repositorio, por lo que necesita entender algunos comandos avanzados que sólo el administrador necesitaría saber en un sistema centralizado.

5.8.1.5. COMPILACIONES CON SCRIPTS

El objetivo de compilar utilizando scripts es automatizar todo el proceso de creación del sistema. Como resultado, el sistema se construye de la misma manera cada vez. Esto elimina los errores ocultos y hace que sea más fácil incorporar nuevos desarrolladores al equipo. También hace que sea más fácil probar y lanzar el software.
Muchas herramientas están disponibles para la creación de compilaciones guionadas. La mayoría de los entornos de desarrollo incluyen una herramienta de compilación, que es lo que generalmente utilizan los equipos de trabajo. Por ejemplo, los proyectos en C y C ++ normalmente utilizan *make*, mientras que los proyectos de Java utilizan Ant. No importa qué sistema utilice el equipo, el objetivo es tener un solo comando o guion (script) que se pueden ejecutar para compilar todo el software. Ese script debe estar bajo un sistema SCM de manera que se pueda verificar el proyecto en cualquier máquina y construir con éxito el sistema.
El problema más común de los equipos a la hora de crear un sistema construido con scripts es omitir los componentes clave. El script de compilación debe establecer todas las variables de entorno necesarias e incluir todos los archivos de datos y las versiones correctas de todas las bibliotecas dependientes en el sistema de administración de código fuente.
También se debe estar alerta para el mantenimiento del sistema de compilación con scripts. Es fácil permitir que los procedimientos manuales se inserten de nuevo en el proceso de compilación, por lo que se deben probar regularmente mediante la construcción en un sistema limpio o una máquina virtual antes de que sean un problema.

5.8.1.6. LA DISCIPLINA EN UN ENTORNO INTEGRADO

El uso de un código base compartido bajo SCM y un sistema de compilación con scripts crea un entorno integrado y promueve un mayor nivel de interacción del equipo. Si el equipo es disciplinado acerca de compartir y coordinar, se puede maximizar el beneficio de estas nuevas herramientas.

5.8.1.7. COMPARTIR

Se debe reunir y compartir tan a menudo como sea posible, preferiblemente luego de completar algún progreso. Si cada integrante del equipo compila en función del código más reciente, se descubrirán rápidamente problemas de integración como bugs y defectos de diseño antes de que consuman más tiempo y sean más costosos de reparar. Comprometiéndose a menudo también puede mejorar la moral del equipo. Cuando cada miembro actualiza con regularidad hace que el progreso sea más tangible y aumenta la sensación de un objetivo compartido, lo que aumentará la productividad de todo el equipo.

5.8.2. PRUEBAS AUTOMATIZADAS

La prueba de errores es un concepto fundamental del desarrollo Lean. Es una parte central de la producción de un producto de calidad y la reducción de residuos (mediante la eliminación del reproceso). Esto es tan cierto en el desarrollo de software como en la manufactura. Las pruebas automatizadas son el principal medio de prevención de errores en el desarrollo de software.
No es coincidencia que las pruebas automatizadas sean una piedra angular de las metodologías agiles, tanto que ya se asume que a nadie consideraría escribir nuevo código sin ellas. Aunque esto no es tan cierto cuando tenemos código heredado, incluso allí la tendencia apunta hacia el aumento del uso de pruebas automatizadas.
El término en si es muy amplio. Aquí lo usaremos para referirnos en general a todo tipo de pruebas: pruebas unitarias, de integración, de aceptación, de especificaciones ejecutables, de rendimiento, de carga, desarrollo guiado por pruebas, desarrollo guiado por comportamiento, etc.
Cada uno de estos diferentes tipos de pruebas tienen un enfoque particular, pero todos tienen en común lo siguiente:

- Las pruebas se crean manualmente por los desarrolladores.
- Se pueden ejecutar de manera automática, sin intervención humana.
- Son ejecutadas por alguna herramienta de prueba.
- Las fallas son detectadas automáticamente.
- El desarrollador es notificado cuando se producen las fallas.

Las pruebas automatizadas apoyan tres de los principios fundamentales del Desarrollo de Software Lean que se describieron en el marco teórico:

- Eliminar desperdicios.
- Construir la calidad.
- Crear el conocimiento.

Esta práctica que abordamos, ayuda a eliminar los residuos del reproceso que se generan cuando los defectos se deslizan a través de las fases posteriores del ciclo de vida del desarrollo de software. Tales defectos son especialmente caros (en tiempo, dinero y reputación) cuando están presentes en todo el camino hasta llegar a la implementación.
Son un método principal para la construcción de la calidad. Un código base con un conjunto completo de pruebas se auto controla y auto valida. Esto ayuda a reducir la probabilidad de que los errores no detectados se introduzcan en el software.
Finalmente, las pruebas automatizadas sirven como documentación "viva" sobre la forma de utilizar realmente las API del código base. Esto crea el conocimiento genuino en el que los desarrolladores confían en realidad, ya que se garantiza su exactitud cada vez que el conjunto de pruebas se ejecuta correctamente.

5.8.2.1. ¿PORQUE PROBAR?

Existen numerosos beneficios al aplicar pruebas, siendo los más básicos la productividad y la calidad del software. La forma más rápida de aumentar ambas es tener un conjunto de pruebas automatizadas.
Cuando un proyecto carece de un conjunto de pruebas, los desarrolladores son muy cautelosos acerca de hacer cambios o adiciones, especialmente si no están familiarizados con el código. Esto significa que se dedica gran cantidad de tiempo a estudiar el código que será modificado y el uso del mismo a través de todo el código base. Incluso entonces, los desarrolladores pueden sentir que pasaron por alto algo importante. Esto añade una gran cantidad de tiempo tanto para el desarrollo de nuevas características y la resolución de problemas.
Si el proyecto cuenta con una serie de pruebas automatizadas, éstas actúan como una red de seguridad para los desarrolladores. En lugar de gastar una cantidad excesiva de tiempo para estudiar y comprender el código global, el desarrollador puede simplemente implementar la función (o arreglar el problema) con una comprensión más simple del código inmediato. Esto se puede hacer de forma segura porque se sabe que cualquier consecuencia adversa imprevista será detectada por las pruebas automatizadas.
Cuando se puede añadir código de manera segura o modificarlo sin una excesiva cantidad de investigación, se puede ahorrar una considerable cantidad de tiempo. Además, dado que los problemas se detectan y corrigen inmediatamente, se puede eliminar el costoso reproceso que se produciría si se hubieran detectado en una fase más avanzada del proyecto.
Por último, cuando un desarrollador quiere saber cómo funciona un segmento de código, por lo general prefiere evitar cualquier documentación detallada porque la experiencia ha demostrado que dicha documentación esta desactualizada y por lo tanto, errónea. Es por esto que los desarrolladores prefieren estudiar el código ellos mismos, lo que les permite documentar efectivamente como se deben utilizar las piezas de código, pudiendo ser probadas y actualizadas simplemente mediante su ejecución. Debido a esto, los desarrolladores llegan a depender de las pruebas, utilizándolas como documentación fiable y funcional.
En resumen, tener un conjunto completo de pruebas automatizadas es una de las cosas más importantes que se puede hacer, ya que éstas:

- Repelen los *bugs.*
- Ayudan a localizar defectos.
- Facilitan los cambios (actúan como una red de seguridad).
- Simplifican la integración.
- Mejoran el diseño del código (lo que facilita los cambios).
- Documentan el comportamiento real del código.

5.8.2.2. TIPOS DE PRUEBAS

5.8.2.2.1. PRUEBAS UNITARIAS

Son el tipo de prueba más básico. Generalmente se organizan con una correspondencia una a una respecto a la organización física del código que se está probando. Al ser una organización fácil de entender, no es sorprendente que éste sea el tipo más popular de pruebas automatizadas.
En un lenguaje orientado a objetos como Java, C++ o C#, las unidades de código que se ponen a prueba son los métodos públicos dentro de una clase. Al igual que una clase recoge un conjunto de métodos relacionados, un banco de pruebas para una clase dada contendrá las pruebas unitarias para los métodos dentro de esa clase. Normalmente habrá un banco de pruebas para cada clase.
Pruebas de Integración
Mientras las pruebas unitarias están diseñadas para controlar unidades individuales de código (en tanto el aislamiento sea posible), las pruebas de integración permiten determinar cómo estas unidades trabajan juntas. A veces esto simplemente significa dejar que el código corra y llame a sus dependencias reales. También podría significar escribir nuevas pruebas que estén diseñadas específicamente para probar las interfaces entre las clases o módulos. En la siguiente figura se ilustran las diferencias básicas entre las pruebas unitarias y de integración.

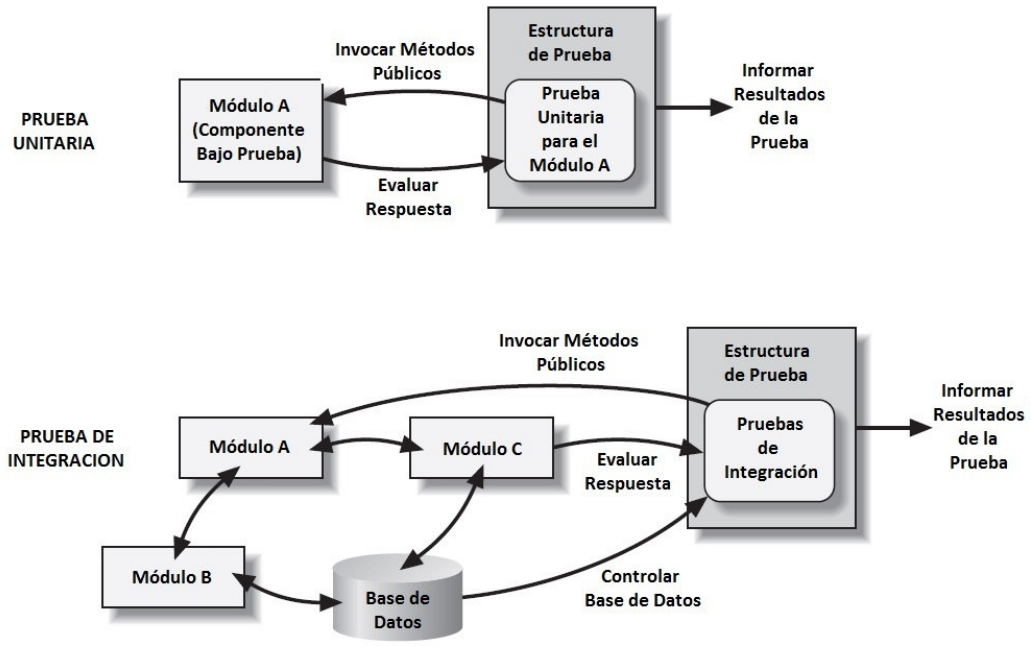


Figura 5.17 Pruebas Unitarias y de Integración

5.8.2.2.2. PRUEBAS DE COMPORTAMIENTO

Son un desarrollo reciente en la escena de las pruebas. Aun así, la mayoría de los defensores de las pruebas automatizadas se están trasladando hacia ellas, dándoles un gran avance en lo que respecta a credibilidad. Las pruebas de comportamiento son a su vez una perspectiva diferente en el tema y un nuevo conjunto de herramientas para expresar directamente esa nueva perspectiva.
Las pruebas de comportamiento no tienen en cuenta la organización física del código o cosas tales como clases o módulos. En su lugar, se centran en las características individuales de la aplicación o sistema y en el comportamiento que la misma debe exhibir para implementar la función. Esos comportamientos son los probados.

5.8.2.2.3. ESPECIFICACIONES EJECUTABLES

Otra tendencia reciente en las pruebas automatizadas ha sido el uso de pruebas ejecutables como especificaciones o requisitos. Esta idea se ha desarrollado a partir de dos campos separados. El primero es el Marco Para Pruebas Integradas (FIT), que utiliza tablas en páginas wiki para especificar las pruebas que se van a ejecutar por la herramienta FIT. La segunda es una extensión natural de las pruebas de comportamiento, pero a un nivel más alto que expresen los requisitos del sistema.
Ambos enfoques de especificaciones ejecutables comparten los mismos atributos: son legibles y comprensibles para los interesados en el negocio de la misma manera que para los desarrolladores de software y son ejecutadas como pruebas por la computadora.

5.8.2.2.4. PRUEBAS NO FUNCIONALES

Hay una serie de áreas en las que una aplicación o sistema se debe controlar que no están relacionadas a sus funciones. Estas son algunos de ellas:

- Pruebas de rendimiento.
- Pruebas de carga.
- Pruebas de seguridad.
- Pruebas de compatibilidad.
- Pruebas de usabilidad.

Este tipo de pruebas son mucho más difíciles de automatizar que los vistos anteriormente. De hecho, algunos de ellos pueden ser imposibles de automatizar satisfactoriamente, como por ejemplo las de usabilidad.
Debido a esto, este tipo de pruebas se han mantenido realizándose de manera manual. Existen herramientas disponibles para ayudar en las pruebas de rendimiento y de carga, pero es raro que estén totalmente automatizadas.

5.8.2.2.5. PRUEBAS DE INTERFAZ DE USUARIO

Independientemente de que se trate de la interfaz de usuario de una aplicación de escritorio o la interfaz del navegador en una aplicación web, estas pruebas son otra área que ha sido

difícil de automatizar. Las herramientas para ayudar la automatización están disponibles, pero tienden a ser específicas de un entorno particular y son difíciles de crear y mantener.
Un punto positivo potencial son las aplicaciones web basadas en un navegador. Desde que el entorno y las tecnologías que subyacen en aplicaciones web están basados en estándares, es posible construir herramientas bastante genéricas para automatizar las pruebas de interfaz de usuario. Una de estas herramientas, llamada *Selenium*, está escrita en JavaScript y se ejecuta en el navegador. Permite escribir pruebas automatizadas que pueden simular las acciones del usuario y comprobar si las respuestas de la aplicación son las correctas.
Debido a que son difíciles de automatizar, una estrategia alternativa tradicional es dividir el código de interfaz de usuario en dos partes. La mayor parte del código implementa la "lógica del negocio", con una fina capa de interfaz gráfica en la parte superior de la representación. Las pruebas automatizadas entonces son dirigidas a la capa de negocio.
No importa que enfoque se elija para realizar las pruebas, en realidad lo más importante es tener un conjunto de pruebas automatizadas y procurar un nivel de cobertura que brinde beneficios que superen con creces los costos.
No es realista, y probablemente no justifique el costo, lograr cubrir al 100% el código fuente. Para código y proyectos nuevos, una cobertura del 60 al 80% es un objetivo razonable, aunque desarrolladores experimentados normalmente buscan una cobertura de 80 a 95%. Existen herramientas que examinan el código fuente y las pruebas y luego determinan cual es el nivel de cobertura (por ejemplo, *Emma* y *Cobertura* para Java y *NCover* para .NET).

5.8.3. INTEGRACION CONTINUA

La integración es donde todo el esfuerzo de desarrollo se une. Los componentes individuales, bases de datos, interfaces de usuario y recursos del sistema están todos unidos entre sí y probados a través de la arquitectura subyacente. La integración supone la verificación de la comunicación adecuada de los componentes a través de interfaces públicas, asegurando que los formatos de datos y contenido del mensaje sean compatibles y completos, asegurándose de que las restricciones de tiempo y recursos se respeten. Los procesos de desarrollo tradicionales tratan a la integración como una fase separada que ocurre después de que todas las piezas se han implementado, siendo típicamente un proceso laborioso que requiere mucho tiempo y que implica una gran cantidad de depuración y revisión. La integración continua, o IC, convierte la fase de integración tradicional en una actividad permanente que se produce durante la ejecución del sistema.
La integración continua no es simplemente una fase repartida a lo largo de todo el ciclo de desarrollo, al menos no como la fase de integración tradicional de desarrollo. La IC es el proceso de integración de pequeños cambios en una base continua con el fin de eliminar la fase de integración por separado antes del lanzamiento del producto. Mediante la integración de los cambios pequeños en intervalos cortos, los desarrolladores pueden hacer crecer el producto de a poco, garantizando que cada nueva pieza trabaja con el resto del sistema.
Esencialmente, la IC es una práctica en la que los desarrolladores integran cambios pequeños mediante el alta de actualizaciones en un sistema de administración de código fuente (SCM) y la construcción del sistema para asegurar que nada se ha roto. Implementar esto puede ser tan simple como construir y probar todo el código fuente o tan complejo como la creación de una versión completamente probada, inspeccionada y entregada.

La integración continua tiene varios requisitos previos:

***Administración de código fuente**:* tema visto en detalle en la práctica 0.
***Marco de prueba de unidades**:* las pruebas automatizadas son una parte integral de cualquier aplicación IC útil, por lo que un marco de pruebas de unidad y un sólido conjunto de pruebas unitarias son necesarios.
***Una compilación automatizada de principio a fin**:* se utiliza una secuencia de comandos para controlar el desarrollo del producto desde el código fuente hasta su lanzamiento.
***Un Servidor de compilación dedicado**:* la integración continua se puede lograr sin una máquina dedicada a la compilación del software, pero es más efectivo contar con un servidor de compilación dedicado. Esto ayuda a eliminar los problemas derivados de los cambios de configuración causados por otras actividades.
***Software de integración continua**:* éste debe detectar cambios en el repositorio de SCM, invocar los scripts que construyen el producto e informar de los resultados de la compilación. El equipo de desarrollo puede escribir software de IC personalizado, pero es más común el uso de una aplicación de terceros.

5.8.3.1. COMPILACIÓN AUTOMATIZADA DE EXTREMO A EXTREMO

Una compilación automatizada de extremo a extremo crea un producto de software completo desde el código fuente en bruto mediante la invocación de todas las acciones necesarias de forma automática y sin intervención por parte de un desarrollador. Implementada a través scripts de compilación, se caracteriza por tres rasgos:

- Empiezan desde cero
- Son procesos de extremo a extremo
- Informan automáticamente los resultados

5.8.3.2. COMPILACIÓN DESDE CERO

La construcción de la IC se inicia mediante la recuperación de una nueva copia del último código fuente desde el repositorio de SCM. Las compilaciones basadas en scripts utilizadas para otros fines solo pueden compilar aquellos módulos que han cambiado, pero la construcción de la Integración Continua siempre inicia desde cero. Reconstruir todo elimina la posibilidad de cambios de dependencia sin ser detectados.
Como excepción a lo expresado podemos señalar que al usar IC en los sistemas grandes y complejos, puede que se tenga que romper lo construido en piezasmás pequeños para reducir la cantidad de tiempo requerido. Fragmentar la construcción en base a las relaciones y dependencias del módulo asegura que el equipo construye y pone a prueba todo el código afectado por un cambio.

5.8.3.3. COMPILAR DE EXTREMO A EXTREMO

La construcción de Integración Continua es un proceso de extremo a extremo que se inicia a partir del código fuente e incluye todas las acciones necesarias para crear el producto de software final. Las definiciones de “todas las acciones necesarias” y “producto de software

final" son arbitrarias y diferentes equipos de desarrollo pueden aplicar distintos significados para cada uno. Por ejemplo, la compilación de extremo a extremo más simple consiste solamente en compilar el código fuente en un archivo ejecutable.
Mientras que una simple compilación de extremo a extremo implementa un proceso repetible para la creación de ejecutables, que no incorpora todos los pasos que normalmente se realizan antes de la liberación de un producto de software. Las construcciones con IC son a menudo más complejas y pueden incluir:

- Prueba unitarias y de integración
- Inicialización de base de datos
- Análisis de cobertura de código
- Inspecciones de Normas de codificación
- Pruebas de rendimiento y carga
- Generación de documentación
- Integrar ejecutables al producto desplegable
- Implementación automática de productos

La siguiente figura muestra una típica compilación de extremo a extremo usada en un proceso de IC, incluyendo la implementación automática del producto. Se debe tener cuidado con este concepto, ya que actualizar un producto mientras está siendo probado o usado puede causar confusión y puede generar disminución de la productividad, pérdida de datos y falta de confianza en el producto.

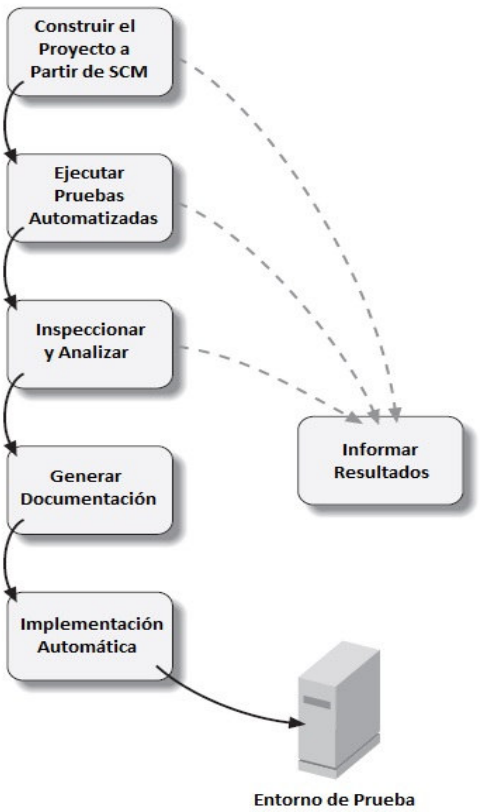


Figura 5.18 Típica Compilación de extremo a extremo

5.8.3.4. INFORME DE LOS RESULTADOS

La Integración Continua emplea mecanismos de información automatizados para asegurar que los desarrolladores, administradores y posiblemente los clientes siempre sepan el estado del proceso de compilación. El contenido del informe depende de la complejidad, una construcción simple puede informar errores de compilación, mientras que compilación más complejas pueden incluir mensajes de error, resultados de las pruebas unitarias y de integración, estadísticas de código, adhesión a las normas y estadísticas generadas por las pruebas de rendimiento.
Una parte importante del proceso está en conseguir los informes de los desarrolladores rápidamente. Las compilaciones segmentadas interrumpen el flujo normal del desarrollo ya que los desarrolladores que trabajan en otras partes del sistema no pueden utilizar la última versión del código fuente de SCM. Cuanto más rápido se detecte la situación antes, más pronto se arreglará y restaurará el flujo.
Los resultados pueden ser reportados en un número de maneras diferentes. Algunos de los métodos más comunes son:

Notificaciones por Correo Electrónico
Construir fracasos genera notificaciones por correo electrónico al equipo de desarrollo, alertando a todos que la compilación está dañada y necesita atención.

Por Páginas web
Las páginas Web son otra manera práctica de mostrar constantemente el estado desarrollo. Muchos de los paquetes de software utilizados para implementar IC incluyen una aplicación web que rastrea y muestra el estado de forma automática, permitiendo que cualquiera pueda ver la situación mediante una simple visualización de la página web.

Los indicadores físicos
Los indicadores físicos, como las luces rojas y verdes, se pueden utilizar para indicar el estado del desarrollo. A pesar de que no proporcionan datos, estos muestran el estado actual de una manera fácil de interpretar.

5.8.3.5. SOFTWARE DE INTEGRACION CONTINUA

Las aplicaciones de software de IC, también conocidos como servidores de IC, deben hacer tres cosas para controlar el proceso: detectar cambios en el repositorio de SCM, invocar scripts de compilación y comunicar los resultados.

5.8.3.6. DETECTAR CAMBIOS EN EL REPOSITORIO SCM

Debido a que la integración continua se basa en la reconstrucción del sistema cada vez que cambia el código base, se requiere la capacidad de detectar estos cambios de forma automática. Los servidores de IC suelen incluir la capacidad para interactuar con una amplia variedad de sistemas SCM, que les permite controlar el repositorio y tomar medidas cuando cambia el código base o aplazar la acción cuando no se detectan cambios.

5.8.3.7. INVOCAR SCRIPTS DE COMPILACIÓN

Una vez que un servidor de IC ha detectado un cambio en el código base, se inicia un proceso automático de creación mediante la invocación de un script de compilación. Normalmente, los servidores de integración continua incorporan un mecanismo para asegurar que todos los cambios pendientes en el repositorio de SCM se han completado antes de invocar el script. Una vez que se ha iniciado la compilación, el servidor de IC supervisa el proceso y espera a que termine.

5.8.3.8. INFORMAR RESULTADOS DE LA COMPILACIÓN

La responsabilidad final de un servidor de IC es dar a conocer los resultados del proceso al equipo de desarrollo y cualquier otra persona que necesita saber el estado de la compilación. Los servidores de integración continua suelen proporcionar notificaciones de error por correo electrónico y acceso vía web a los últimos resultados de compilación.

5.8.3.9. LOS SERVIDORES DE IC PUEDEN PROGRAMAR UN CRONOGRAMA DE COMPILACIÓN

Una interpretación estricta de la integración continua no incluye compilaciones programadas, pero estas pueden ser útiles en el manejo de productos grandes y complejos. Usando una combinación de compilaciones disparadas por cambios en el código y compilaciones programadas, los desarrolladores pueden crear un sistema que realiza pruebas unitarias y de integración rápidas para todos los cambios en el código y pruebas más largas de fondo. La mayoría de los servidores de IC permiten programar compilaciones para un momento determinado del día (durante la noche, por ejemplo) o de forma periódica.

5.8.3.10. IMPLEMENTACIÓN DE LA INTEGRACIÓN CONTINUA

Como hemos mencionado al principio del desarrollo de esta práctica, integración continua es el proceso de integrar continuamente pequeños cambios de código en un producto de software. Cada vez que cambia el código base (es decir que los cambios son incluidos en el repositorio de SCM), el sistema está compilado y probado. Los desarrolladores son inmediatamente notificados de cualquier error causado por los cambios, lo cual les permite hacer correcciones rápidamente. La IC puede considerarse como una extensión de las construcciones automatizadas en el que se realiza una compilación automatizada (incluyendo las pruebas, informes y otras acciones) cada vez que cambia el código base.
Si se ha decidido implementar IC en un proyecto, se debe comenzar por cumplir con tres requisitos previos indicados al comienzo del desarrollo de la práctica:

- Establecer un repositorio de SCM para todo el código fuente, scripts, datos de prueba, etc.
- Identificar un servidor de compilación. Asegurarse de que el servidor tenga acceso al repositorio de SCM.
- Identificar el o los marcos de pruebas unitarias necesarios para apoyar las pruebas del proyecto. Cargar la estructura del software en el servidor de compilación.

Si ya se ha implementado un compilador automatizado utilizando un lenguaje basado en scripts, el siguiente paso ya está hecho. Si no es así, se debe trabajar a través de la siguiente lista para establecer un script de compilación automatizado para el proyecto:

- Elegir un lenguaje de compilación basado en scripts que soporte el software de SCM y el marco de pruebas unitarias que se va a utilizar.
- Cargar el software de secuencias de comandos en el servidor de compilación.
- Crear un script de compilación sencillo que recupere todo el código fuente desde el repositorio de SCM.
- Añadir una tarea de compilación al script de cada módulo en el proyecto.
- Añadir el script de compilación al repositorio de SCM.

Con un script de compilación para cada módulo en su lugar, es el momento de crear el proceso de integración continua propiamente dicho. Asumiremos que se va a utilizar uno de los servidores de IC disponibles:

- Seleccionar un servidor de IC con las características y compatibilidades necesarias (SCM, lenguaje de compilación con scripts y un marco de pruebas unitarias).
- Cargar software servidor de IC en la máquina de compilación.
- Configurar el servidor de IC para controlar el repositorio de SCM del proyecto y ejecutar el script de compilación principal cuando se detecten cambios.
- Iniciar el servidor de integración continua.

Virtualmente todos los servidores de IC tienen la capacidad de controlar los cambios en el repositorio, por lo que una vez iniciado, el mismo debe iniciar una compilación cada vez que se realicen cambios en el repositorio de SCM.

Una vez que el servidor de IC está compilando el proyecto, se deben informar los resultados. El siguiente paso es añadir un mecanismo de reporte para el proceso de IC. Los resultados se informan por lo general mediante la adición de tareas al script de compilación. Estas pueden generar datos en un formato particular para su visualización en una aplicación web, generar correos electrónicos a los desarrolladores cuando algo falla o establecer el estado de un indicador visual. Se debe seleccionar el mecanismo de información deseado y añadirlo como apropiado para el servidor de IC en uso.

El proceso de integración continua está ahora a punto de hacer algo útil. Todo lo que queda es añadir pruebas unitarias e informar los resultados. Al igual que con los mecanismos de información, las pruebas unitarias normalmente se invocan dentro del script de compilación. La modificación del script para invocar todas las pruebas unitarias existentes hace que sea más fácil añadir módulos posteriormente. Nuevas pruebas unitarias serán recogidas automáticamente a medida que se añadan al repositorio de SCM.

En este punto, el proyecto cuenta con un proceso de IC útil y funcional que compilará el proyecto cuando se detecten cambios en el repositorio y notificará a los desarrolladores si la compilación o las pruebas unitarias fallan. Se puede mejorar el proceso de IC para incluir otras tareas como el análisis de código, pruebas de carga, documentación e implementación del producto mediante la adición de la tarea apropiada al script de compilación.

5.8.3.11. LOS DESARROLLADORES Y EL PROCESO DE INTEGRACIÓN CONTINUA

Un proceso de IC efectivo, además de una colección de herramientas y scripts, requiere la aceptación de los miembros del equipo de desarrollo. La integración continua trabaja mediante la unión de pequeñas piezas de funcionalidad a intervalos regulares. Los desarrolladores deben cumplir con algunas reglas para que el proceso de IC sea efectivo:

Registros Frecuentes
Los desarrolladores deben comprobar el código en el repositorio deSCM al menos una vez al día, (preferiblemente varias veces al día). Cuanto más tiempo el código modificado permanezca sin registrar, es más probable que no esté sincronizado con el resto del proyecto, lo que lleva a problemas de integración cuando finalmente se registre.

Sólo Registrar Código Funcional
Aunque el código se puede comprobar con frecuencia, no debe ser registrado si aún no es funcional, no se compila o fallan las pruebas unitarias. Los desarrolladores deben compilar el proyecto actualizado de forma local y ejecutar las pruebas unitarias aplicables antes de registrar el código en el repositorio.

La tarea de Más Alta Prioridad es Arreglar Compilaciones con fallas
Cuando se detecta una falla en la compilación, arreglarla se convierte en la más alta prioridad. Permitir que una falla de compilación persista aumenta el riesgo de problemas de integración para el desarrollo en curso, por lo que los desarrolladores necesitan centrarse en arreglar el problema antes de seguir adelante.

5.8.3.12. CONSTRUCCIÓN DE LA CALIDAD A PARTIR DE LA INTEGRACIÓN CONTINUA

La integración continua beneficia a una organización de desarrollo de software de varias maneras. A continuación se describen varios puntos asumiendo la implementación de IC, incluyendo un robusto conjunto de pruebas unitarias y de integración que se ejecutan cada vez que se vuelve a reconstruir el código fuente.

5.8.3.12.1. Ayuda a Depurar al Limitar el Alcance de los Errores

Una implementación efectiva de IC requiere que los usuarios comprueben los cambios en el repositorio de SCM periódicamente. Frecuentes registros de entrada tienden a limitar el tamaño y el alcance de los cambios, que a su vez limitan las posibles fuentes de error cuando una prueba falla. La integración continua ayuda a depurar velozmente al limitar el origen de un error a los cambios recientes.

5.8.3.12.2. Proporciona Retroalimentación para Realizar Cambios

La retroalimentación para los cambios no se limita a los fallos de las pruebas; incluye integraciones exitosas también. La integración continua proporciona retroalimentación tanto positiva como negativa sobre los cambios recientes, permitiendo a los desarrolladores ver

los efectos de esos cambios rápidamente. La calidad negativa o positiva de estos efectos puede entonces conducir a cambios adicionales o a la determinación de que no requiere más trabajo.

5.8.3.12.3. Minimiza el Esfuerzo de Integración

Los ciclos de vida de desarrollo de software tradicionales incluyen una fase de integración entre el final del desarrollo y el lanzamiento del producto. La integración continua minimiza (y a veces elimina completamente) la fase de integración. Al repartir el esfuerzo de integración a lo largo de todo el ciclo de desarrollo, la IC permite a los desarrolladores integrar cambios más pequeños. Esto con frecuencia no significa simplemente difundir el mismo esfuerzo a lo largo de un período mayor; en realidad reduce el esfuerzo de integración. Debido a que los cambios son pequeños y se limitan a secciones específicas del código fuente, el aislamiento de los errores es más rápido y la depuración es más fácil.

5.8.3.12.4. Minimiza la Propagación de Defectos

Un defecto no detectado en un módulo puede propagarse a través del sistema de software a través de dependencias con otros módulos. Cuando el defecto es finalmente descubierto, la solución no sólo afecta al módulo defectuoso, sino también todos los módulos que dependen de él. En primer lugar, el defecto se propaga a través del sistema y luego la solución se propaga a través del sistema. La integración continua minimiza este problema mediante la detección del defecto a tiempo (a través de las pruebas automatizadas unitarias y de integración) antes de que tenga la oportunidad de propagarse.

5.8.3.12.5. Crea una Red de Seguridad para los Desarrolladores

Una parte importante de cualquier esfuerzo de desarrollo implica modificar el código existente para tener en cuenta los requisitos cambiantes y módulos de refactorización para mejorar la implementación del software en general. Los desarrolladores pueden modificar y refactorizar el código con confianza ya la integración continua proporciona una red de seguridad. Las pruebas unitarias y de integración que se ejecutan como parte de la construcción dan retroalimentación inmediata a los desarrolladores, quienes pueden deshacer los cambios si algo causa un quiebre en la compilación.

5.8.3.12.6. Asegura que el Mejor y Más Nuevo Software Esté Siempre Disponible

Debido a que la IC compila el proyecto cada vez que cambia el código, se mantiene la versión más reciente en sincronía con el repositorio de código. Una implementación de IC que lee desde un repositorio de SCM y despliega la compilación en un entorno de prueba asegura que los desarrolladores, probadores y usuarios están trabajando con la última versión del software.

5.8.3.12.7. Proporciona una Imagen Actual del Estado del Proyecto

Debido a que un proceso de integración continua informa el estado de cada compilación, se proporciona una visión actual del estado del proyecto. La "fotografía" puede incluir:

Los resultados de la última compilación, la cual le dice a los desarrolladores lo que necesita ser arreglado antes de seguir adelante.
Las métricas resultantes de las pruebas unitarias, cobertura de código y el análisis de los estándares de codificación, lo que permite a los líderes del equipo medir la adherencia a los procesos.
Número de pruebas aprobadas, lo que indica cuantos requerimientos se han completado en realidad y puede permitir que los directivos conozcan el progreso del proyecto.

5.8.4. MENOS CODIGO

El título de esta práctica no se refiere a escribir menos software; sino a realizar el trabajo con menos líneas de código. Se trata de eliminar código innecesario y hacer que el necesario sea más eficiente, intentando que se mantenga pequeño sin dejar de ofrecer funcionalidad (y por lo tanto valor) al cliente.
El tamaño del código base afecta a los proyectos de varias maneras, siendo la más evidente la cantidad de tiempo necesario para escribirlo. Sin embargo, un código de gran tamaño ha alcanzado efectos más amplios:

Por lo general significa más componentes, que a su vez implica más conexiones entre esos componentes y un mayor esfuerzo de integración.
Conlleva más errores, lo que significa mayor depuración.
Son más difíciles de entender, aumentando la curva de aprendizaje para los nuevos desarrolladores y para los futuros esfuerzos de mantenimiento.
Poseen mayor "inercia", lo que dificulta su respuesta al cambio.

A medida que grandes códigos bases manejan más componentes, se generan más errores y curvas de aprendizaje más largas que elevan los costos de desarrollo y mantenimiento.
La definición de residuo en el desarrollo Lean es cualquier cosa que aumenta los costos sin aportar valor, por lo que el costo de desarrollo y mantenimiento de códigos base innecesariamente grandes puede ser visto como un residuo.
Escribir menos código obliga a los desarrolladores a adoptar una actitud que mira críticamente cada línea escrita. Sin embargo, reducir al mínimo el tamaño de un código base no se limita a la implementación. Todos los aspectos del desarrollo afectan la cantidad de líneas escritas, por lo que los desarrolladores deben aplicar una actitud minimalista a lo largo de todo el proceso de desarrollo.
Los métodos tradicionales de desarrollo de software animan a los clientes a crear conjuntos completos de requisitos iniciales. Cuando se les aclara que es más costoso hacer cambios en los requerimientos una vez avanzado el ciclo de desarrollo, los clientes crean requisitos para todo lo que se les ocurra. La eliminación de los requisitos que impulsan el desarrollo de funciones no utilizadas puede reducir significativamente el tamaño del código base.

En lugar de basarse en este enfoque centrado en la especulación en vez de las necesidades conocidas y la creación de funciones no utilizadas, los requisitos deberán ser construidos con el tiempo. Los mismos deben basarse en las necesidades reales y no en la imaginación. La lista completa de requisitos puede ser creada y refinada de manera incremental a medida que los clientes y los desarrolladores obtienen una mejor comprensión del problema y la solución.
Diseños que sean lo suficientemente flexibles para adaptarse a los cambios y adiciones futuras pueden generar grandes ahorros de tiempo, pero los diseños que toman en cuenta todas las posibilidades conducen a código no utilizado e interfaces sobrecargadas. Los diseños excesivamente flexibles se centran en la especulación en lugar de lo que realmente se necesita para proporcionar valor inmediato. La negociación inicial para determinar el diseño emergente (que permite al diseño evolucionar para tener en cuenta las necesidades emergentes) conduce a sistemas que son flexibles y eficientes a la vez.
Un conjunto completo de pruebas unitarias es fundamental para el proceso de desarrollo, las cuales añaden valor mediante la construcción de la calidad en el sistema y la reducción de costos de mantenimiento a largo plazo. Sin embargo, desarrollar pruebas unitarias para código innecesario es acumular aúnmás desperdicio. Al igual que cualquier segmento del código base, se requiere tiempo y esfuerzo para escribir, depurar, mantener y ejecutar pruebas unitarias, y en ningún caso ese tiempo y esfuerzo aportan valor añadido al producto final cuando el código es innecesario en primera instancia.
La distorsión del alcance es un fenómeno común en el desarrollo de software. Los clientes cuentan con la posibilidad de cambiar de opinión y pensar en nuevas características, mientras que los desarrolladores tienen una tendencia natural a anticiparse a los cambios futuros y planificar en función a ellos. Los nuevos requisitos son inevitables y es deseable contar con diseños flexibles, pero si se producen sin control pueden conducir a interfaces sobrecargadas y código innecesario.
Haciendo una analogía, una persona fuera de forma y con exceso de peso que desea volver a ser una persona sana, básicamente debe hacer tres cosas: dieta, ponerse en forma y mantener hábitos saludables a través del tiempo. Lo mismo sucede con cualquier código base, como se muestra en la figura 12-5. Para recortar su tamaño y mantenerlo pequeño, tenemos que eliminar el código innecesario, emplear buenas prácticas de codificación y justificar cualquier nuevo código.

5.8.4.1. ELIMINAR CÓDIGO INNECESARIO

La manera más fácil de reducir el tamaño de un código base es eliminar el código innecesario. Cualquier línea o segmento que no sea directamente compatible con los requisitos existentes se considera innecesario, al igual que las pruebas unitarias para ese código. Los ejemplos incluyen interfaces de clases con múltiples métodos sobrecargados, el uso excesivo o la rápida aplicación de patrones de diseño y tener funciones que no ha solicitado el cliente.
Por supuesto, hay momentos en los que anticipar necesidades futuras es una parte integral del software que está siendo desarrollado o que quitar código no utilizado de una aplicación heredada implique más esfuerzo del que vale la pena. Del mismo modo en que una dieta puede lograr el objetivo de perder peso hasta el punto de la desnutrición, la eliminación de código puede ser llevada demasiado lejos. Se debe combinar un análisis cuidadoso con el

sentido común y una visión razonable del futuro del sistema para determinar que código es innecesario.

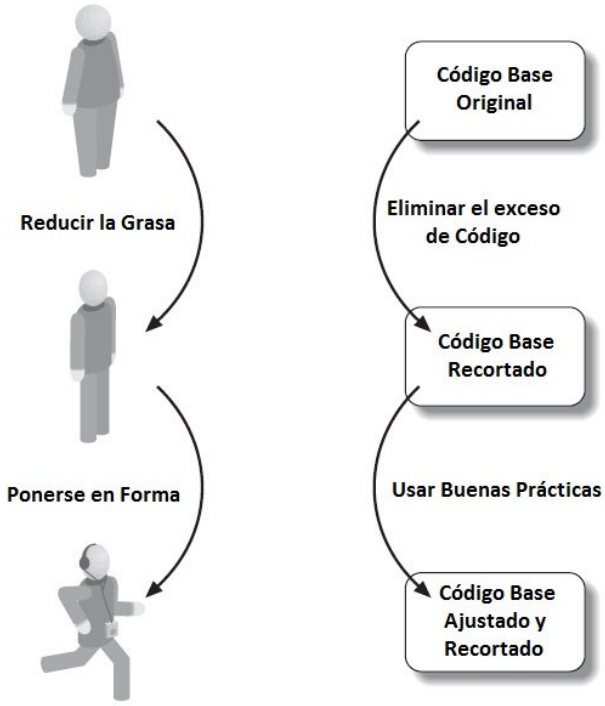


Figura 5.19 Escribir menos código es un programa de acondicionamiento físico

Llevar a cabo una campaña para eliminar todas las partes innecesarias de un código base existente puede tener un efecto perjudicial sobre el desarrollo a continuar. Hay dos enfoques que se pueden utilizar para hacer recortes a través del tiempo y minimizar este efecto. El primero es similar a añadir pruebas unitarias a código heredado: aplicar principios de "menos código" cuando se lo cambia por alguna otra razón, como la corrección de un defecto o cambiar la funcionalidad existente. El segundo enfoque es eliminar código innecesario solo cuando el costo de mantenerlo en el futuro excede el costo de volver a escribirlo ahora.

5.8.4.2. EMPLEAR BUENAS PRÁCTICAS DE CODIFICACIÓN

El filósofo de la administración Peter F. Drucker sostuvo que "***no hay nada tan inútil como hacer eficientemente algo que no debería haberse hecho en absoluto***". Dicho de otra manera, el esfuerzo eficiente es un pobre sustituto de eliminar por completo el esfuerzo, pero no es una excusa para ignorar la eficiencia. Ambos son necesarios para lograr un código base realmente mínimo. Las buenas prácticas de codificación perfeccionan el código, haciéndolo más pequeño, eficiente y comprensible.

5.8.4.3. JUSTIFICAR TODO EL CÓDIGO NUEVO

Una vez que el código está recortado y ajustado, se debe seguir conservándolo así. El mantenimiento de un código base pequeño requiere la aplicación de los dos principios

aplicables a cualquier código nuevo. Solo añadir nuevas líneas si satisface directamente las necesidades actuales y asegurar que se adhiere a las mejores prácticas definidas. Segmentos que existen únicamente para soportar los cambios o funciones previstas no hacen a un código "liviano"; por el contrario, lo hacen "hinchado" e ineficiente. Perder peso y ponerse en forma son solo la mitad de la batalla; mantener el peso y mantenerse en forma requieren diligencia durante toda la vida.

5.8.4.4. DESARROLLAR MENOS CÓDIGO

Cualquier principio o técnica para realizar códigos más pequeños se deben aplicar desde el principio de un proyecto. Mantener pequeño un código base es mucho más fácil que reducir el tamaño de uno existente.
Existen varias técnicas para la aplicación del principio de "menos código". Priorizar requerimientos, desarrollar en iteraciones cortas, desarrollar solo para la iteración actual, evitar la complejidad innecesaria y la reutilización pueden ayudar a eliminar código innecesario. Normas de codificación, las mejores prácticas, patrones de diseño y refactorización pueden ayudar a hacer el código más eficiente.

5.8.4.5. PRIORIZAR REQUISITOS

Una concepto general que a menudo se aplica a los requerimientos del cliente es la regla "80/20" (mencionado al principio de la sección), que establece que:

"80% de la funcionalidad útil de una aplicación de software es descrita por el 20% de los requerimientos"

Mediante la implementación del 20% más significativo de los requerimientos, un equipo de desarrollo puede aplicar el 80% de la funcionalidad requerida por el cliente. Priorizar los requisitos deja en claro que características constituyen el 20% y permite a los desarrolladores concentrarse en ellas.
Los requisitos a menudo se escriben para incluir todo lo que un cliente puede pensar, a fin de cuentas siempre se les ha dicho que es más costoso hacer cambios con el proyecto avanzado que especificar cosas por adelantado. El ochenta por ciento de estos requisitos serán funciones que proporcionan poco valor para el usuario final. Si el equipo de desarrollo trata todos los requisitos de la misma manera, se escribirá una gran cantidad de código para implementar características que al cliente realmente no le preocupan. En otras palabras, una gran cantidad de líneas innecesarias pueden ser añadidas al código base.
¿De qué manera priorizar minimiza el tamaño del código? Cuando se combina con iteraciones cortas y frecuentes, priorizar conduce al equipo a desarrollar temprano las características más importantes y le da al cliente la oportunidad de detener el desarrollo cuando el código existente es "suficientemente bueno" o las características se consideran innecesarias. El código para implementar las funciones no utilizadas nunca se escribe.
De manera más general, los requisitos priorizados son un tema recurrente en el desarrollo de software Lean. Aseguran que los desarrolladores están creando el producto adecuado, proporcionando una hoja de ruta de las características consideradas más importantes por parte del cliente. También juegan un papel en la reducción de residuos, ya que las características de mayor prioridad probablemente sean las que el cliente entiende mejor y es

menos factible que las cambie, por lo que al implementarlas primero se minimiza el riesgo de que se desperdicie esfuerzo de desarrollo.
La eficacia de la priorización está basada en la condición dinámica de las prioridades. La retroalimentación de los clientes a menudo lleva a cambios en sus necesidades que toman la forma de requisitos añadidos, eliminados y modificados. Además, como el producto comienza a tomar forma, los clientes obtienen una mejor comprensión de lo que el producto debe hacer y cómo utilizarlo en el entorno implementado. El cambio de forma del producto cambia la importancia de las funciones específicas y esos cambios se deben reflejar en la priorización de requisitos.
Se deben tener en cuenta varios conceptos clave a la hora de priorizar requisitos:

Basar las prioridades en función del valor al cliente. Esto significa que debe estar involucrado en el proceso. Los desarrolladores no deben dar prioridad a los requisitos de forma directa. Aunque las previsiones sobre una determinada característica pueden influir en el cliente, es él quien decide lo que es más importante.
Volver a evaluar las prioridades a menudo. Se debe comenzar cada iteración con una lista de prioridades actualizada para asegurar que se incorporan los cambios en las necesidades del cliente.
Los requisitos deben ser priorizados de forma individual, no simplemente clasificarlos como de “alta”, “media” o “baja” prioridad. Si se tienen prioridades numéricas únicas se fuerza al cliente a evaluar objetivamente sus necesidades y se eliminan ambigüedades a la hora de seleccionar requerimientos para incluir en una iteración.

5.8.4.6. DESARROLLAR EN ITERACIONES CORTAS

Son un sello distintivo de las metodologías agiles de desarrollo y apoyan directamente los principios Lean sobre la eliminación de desperdicio, aplazando el compromiso y entregando rápido. Las iteraciones cortas proporcionan nuevas funcionalidades a los clientes de forma rápida y con frecuencia, acortando el ciclo de entrega y retroalimentación y reduciendo el tiempo que los desarrolladores y los clientes esperan unos a otros.

5.8.4.7. DESARROLLAR SOLO PARA LA ITERACIÓN ACTUAL

Enfocarse solo en la iteración actual, que utiliza una lista priorizada de requisitos e iteraciones cortas, ayuda a minimizar el código base, centrándose en el desarrollo de los requisitos que son más importantes para el cliente. Suponiendo esto, existen criterios para la implementación de un aspecto del diseño o de una función: solo desarrollar código que apoye directamente los requerimientos asignados a la iteración actual. Mantener este enfoque asegura que las características de mayor prioridad se implementan y que los desarrolladores no se desvíen por el agregado de funciones adicionales. En esencia, desarrollar solo para la iteración actual significa que “si no se necesita en este momento, no lo haga”.

5.8.4.8. REUTILIZAR SOFTWARE EXISTENTE

Una de las maneras más eficaces de escribir menos código es volver a utilizar software existente; maximiza la funcionalidad y reduce al mínimo el tamaño del código base. También

se ocupa directamente de cada uno de los problemas inherentes en los códigos de gran tamaño, ya que la reutilización reduce:

Tiempo de codificación por la disminución de la cantidad de código escrito.
El número de componentes, aunque no necesariamente el número de conexiones entre los componentes.
Depuración mediante la adición de funcionalidades que ya usan código depurado.
Curvas de aprendizaje mediante la limitación del número de diferentes componentes que un desarrollador debe entender.

La reutilización de software existe en muchas formas diferentes, cada una de las cuales afecta al tamaño del código de manera diferente:

Copiar el código fuente de un componente a otro reduce el tiempo de codificación y depuración, pero en realidad aumenta el tamaño del código base.
Los componentes modulares y plug-ins permiten escribir código una vez y volver a utilizarlo varias veces para reducir la cantidad de código fuente.
Servicios que están alojados en un sistema de arquitectura orientada a servicios que minimizan el tamaño del código, permitiendo a los desarrolladores utilizar la funcionalidad existente.
Las bibliotecas y los marcos proporcionan funcionalidad integrada.

Aunque hay que tener en cuenta cuestiones como la aplicabilidad de los componentes existentes, la disponibilidad a largo plazo de los servicios y derechos de licencia y distribución de software de terceros, reutilizar software existente es un enfoque válido para escribir menos código.

5.8.4.9. USAR ESTÁNDARES DE CODIFICACIÓN Y MEJORES PRÁCTICAS

Las normas de codificación ayudan a los desarrolladores a crear un código de fácil comprensión. Se acorta la curva de aprendizaje para los encargados de trabajar en un código que no han escrito ellos mismos y permiten que los desarrolladores pasen menos tiempo averiguando las peculiaridades del estilo de codificación de otros y más tiempo para averiguar lo que hace el código.
Cada lenguaje de programación y entorno de desarrollo tiene su propio estándar de codificación, teniendo la mayoría más de uno. Aunque algunas de las reglas de una determinada norma son arbitrarias y reflejan las preferencias personales del autor, muchas reglas son consistentes de estándar a estándar. Por otra parte, las normas pueden ser modificadas para adaptarse a un equipo de desarrollo o proyecto en particular. Al final, utilizar un estándar de codificación es más importante que el estándar que se elige.
Las mejores prácticas toman el concepto de estándares de codificación un paso más allá y pueden ayudar a evitar errores comunes y reducir la complejidad. Dan a los desarrolladores una guía sobre cómo usar correcto de las características del lenguaje y al igual que las normas de codificación, ayudan a entender el código creado por otra persona.
Una línea borrosa separa a los estándares de codificación de las buenas prácticas, pero ambas apoyan los principios de “menos código”. Crean un lenguaje común (por encima y

más allá del lenguaje de programación) que permite a los desarrolladores entender el código de cada uno y reducen la complejidad al describir la mejor manera de implementar construcciones de código específicas.
Existen aplicaciones de análisis para estándares de codificación y mejores prácticas para muchos lenguajes de programación y entornos de desarrollo. Estas aplicaciones se pueden combinar con el concepto de integración continua para hacer cumplir de forma automática las normas y prácticas elegidas.

5.8.4.10. USAR PATRONES DE DISEÑO

Los patrones de diseño son soluciones generales a problemas recurrentes. Resuelven problemas comunes en diversos contextos mediante la reutilización de la estructura general de una solución conocida y alterándola según sea necesario para adaptarse a la situación. En un sentido, los patrones son estándares y mejores prácticas aplicadas al diseño.
Los patrones se centran en la mejora del diseño de un sistema; sin embargo, su utilización en la implementación de soluciones genera varios efectos secundarios que reducen el tamaño del código base, tales como la eliminación de la duplicación, simplificar el diseño, el uso de funciones avanzadas del lenguaje para implementar interacciones entre los objetos de firma implícita y la creación de componentes reutilizables.
Aunque los patrones de diseño pueden ser apropiados cuando se utiliza el diseño en cantidades limitadas, son más eficaces en el desarrollo Lean como una manera de evolucionar un diseño a través del tiempo. Esto se produce a medida que se añade funcionalidad a través de varias iteraciones y el diseño esta readaptado para acomodarlo. A medida que la aplicación crece y los elementos de los patrones emergen, se puede refactorizar el diseño como instancias de estos patrones emergentes.

5.8.4.11. REFACTORIZAR EL CÓDIGO Y EL DISEÑO

Es el proceso de modificar la implementación interna de un componente sin cambiar su interfaz pública o comportamiento. En el contexto de escribir menos código, el objetivo de la refactorización es reducir la cantidad de líneas mediante la modificación de la aplicación para tomar ventaja de las soluciones conocidas. Para los problemas comunes, las soluciones conocidas por lo general se han ido perfeccionando con el tiempo hasta convertirse en una implementación eficiente, ya sea a nivel de codificación básica o a nivel de diseño.
Como dijimos, refactorizar puede ayudar a reducir el tamaño del código base mediante el aprovechamiento de las soluciones conocidas. Sin embargo, para ser eficaz debe ser parte regular de los trabajos de implementación. Los desarrolladores individuales deben hacer parte de su rutina diaria la refactorización a nivel de código: en primer lugar, implementar un requerimiento como una solución simple, incluso forzándolo, y a continuación refactorizar la solución para que sea lo más eficiente posible. A nivel de diseño, refactorizar requiere la coordinación entre los desarrolladores a medida que los elementos de los patrones emergen. A menudo, un desarrollador sénior asume el papel de arquitecto, revisando la implementación en busca de diseños emergentes sobre una base regular.

5.8.4.12. RESISTENCIA A TENER MENOS CÓDIGO

Es posible toparse con cierta resistencia al intentar implementar las prácticas descritas hasta el momento, sobre todo de los miembros más experimentados de un equipo de desarrollo. La aplicación de las mismas por lo general requiere que los desarrolladores cambien su forma de escribir código.
Cuando los miembros del equipo están acostumbrados a escribir su propio código, a menudo desarrollan desconfianza del software de terceros. La superación de esa actitud puede ser difícil, especialmente con los desarrolladores más experimentados que han tenido malas experiencias con este tipo de software en el pasado. Además, las bibliotecas y marcos pueden venir con licencias y costos de distribución, haciéndolos una opción inicial más costosa.
Los estándares de codificación y mejores prácticas permiten que los desarrolladores cambien la forma en que escriben código. El mejor enfoque puede basarse en escoger uno de los estándares de la industria para el lenguaje y el entorno de desarrollo. Toda modificación a esa norma debe ser acordada con el equipo completo.
Refactorizar de manera eficaz el código y diseño requiere de conocimiento y experiencia, aunque en realidad esto no es una razón válida para evitarlo. En este sentido, una de las maneras de mejorar las habilidades de los desarrolladores es fomentar dicha práctica. Las sesiones de práctica deben llevarse a cabo fuera del tiempo de desarrollo real, lo que conlleva mayor esfuerzo, pero totalmente superado por los beneficios a obtener. Por otro lado, utilizar patrones de diseño puede hacer que el código sea más eficiente, pero el mal uso de dicha herramienta puede aumentar innecesariamente la complejidad del software.

5.8.5. ITERACIONES CORTAS

El desarrollo iterativo ofrece software funcional al cliente para su evaluación en intervalos específicos. Cada iteración añade nueva funcionalidad al producto y aumenta la comprensión del cliente sobre cómo el producto final va a cumplir con sus necesidades. También proporciona una oportunidad para identificar de qué manera el producto puede fallar al cumplirlas. La eficacia del desarrollo iterativo proviene de estas oportunidades para recibir e incorporar la retroalimentación del cliente. El feedback del cliente impulsa la siguiente iteración mediante la adición, supresión y modificación de los requisitos para corregir deficiencias y la longitud de cada iteración determina el número de oportunidades disponibles durante un trabajo de desarrollo. Esto significa que las iteraciones cortas proporcionan una mayor retroalimentación y aumentan la posibilidad de que el producto final sea lo que desea el cliente.
Las iteraciones cortas soportan tres de los principios del Desarrollo Lean, los cuales son:

- Eliminar residuos
- Retrasar el compromiso
- Entregar rápido

Las dos formas de residuos que son frecuentes en el desarrollo de software son el trabajo parcialmente terminado y la replanificación. El trabajo parcialmente terminado es cualquier tarea que se ha iniciado pero no se ha completado; incluye cosas tales como requisitos no codificados y código incompleto y no probado. Dado que el trabajo parcialmente terminado

no contribuye una nueva funcionalidad al producto actual, es un trabajo que no aporta valor agregado. Además, si el cliente decide detener el desarrollo, en un área específica o para todo el producto, cualquier trabajo parcial es un esfuerzo desperdiciado. La planificación demasiado lejos en el futuro puede resultar en la replanificación cuando las necesidades del cliente cambian. El esfuerzo empleado en la planificación inicial se pierde cuando el trabajo se vuelve a planificar. Las iteraciones cortas previenen ambas situaciones, centrando esfuerzos en el rápido desarrollo los de requisitos del cliente de mayor prioridad y planificar con anticipación solo lo suficiente para apoyar ese desarrollo.
El aplazamiento del compromiso significa aplazar las decisiones irrevocables el mayor tiempo posible (y razonable). El retraso de este tipo de decisiones ofrece a los desarrolladores tiempo para reunir la información necesaria para decidir y las iteraciones cortas proporcionan oportunidades para recoger esa información en forma de comentarios de los clientes. Los desarrolladores pueden crear prototipos de la aplicación final en una iteración inicial, intercambiar ideas y el usar la retroalimentación para decidir sobre la implementación final en una iteración posterior.
Como se señaló anteriormente, el desarrollo iterativo pone nueva funcionalidad en las manos del cliente a intervalos específicos. Las iteraciones cortas apoyan el principio de "rápida entrega" mediante la reducción de tamaño del intervalo y la entrega de nueva funcionalidad como una respuesta rápida a la retroalimentación de los clientes. En lugar de esperar hasta la entrega final del producto para ver los efectos de sus comentarios, los clientes lo ven en la siguiente iteración. El uso de iteraciones cortas en el Desarrollo de Software Lean se basa en dos principios fundamentales y técnicas específicas para la aplicación de esos principios.

5.8.5.1. LAS ITERACIONES CORTAS GENERAN VALOR PARA EL CLIENTE

Dos principios importantes conducen al uso de iteraciones cortas en el Desarrollo de Software Lean: aumento de las oportunidades de retroalimentación y hacer correcciones sobre la marcha. Obtener feedback por parte del cliente e implementarlo permite desarrollar el producto de manera eficiente, entregar el producto rápidamente y asegurarse de que el producto final es compatible con las necesidades del cliente.

5.8.5.2. AUMENTAR LAS OPORTUNIDADES DE RETROALIMENTACIÓN

Uno de los objetivos de iteraciones cortas es aumentar la cantidad de información recibida de los clientes. A menudo las iteraciones cortas proporcionan nuevas funcionalidades, lo que aumenta el número de oportunidades que tienen los clientes para evaluar el producto. La Figura 12-6 muestra gráficamente cómo iteraciones más cortas aumentan las oportunidades de feedback. Un esfuerzo de desarrollo de 12 meses que consiste en una sola iteración no proporciona ninguna oportunidad para obtener retroalimentación hasta que el producto final es entregado, momento en el que ya es demasiado tarde. Dividiendo el esfuerzo en dos iteraciones se proporciona alguna información, permitiendo a los desarrolladores modificar el software creado durante los primeros seis meses en los segundos seis meses. Si fragmentamos el desarrollo en iteraciones mensuales, el equipo puede obtener feedback de los clientes en 11 puntos durante el esfuerzo de desarrollo.

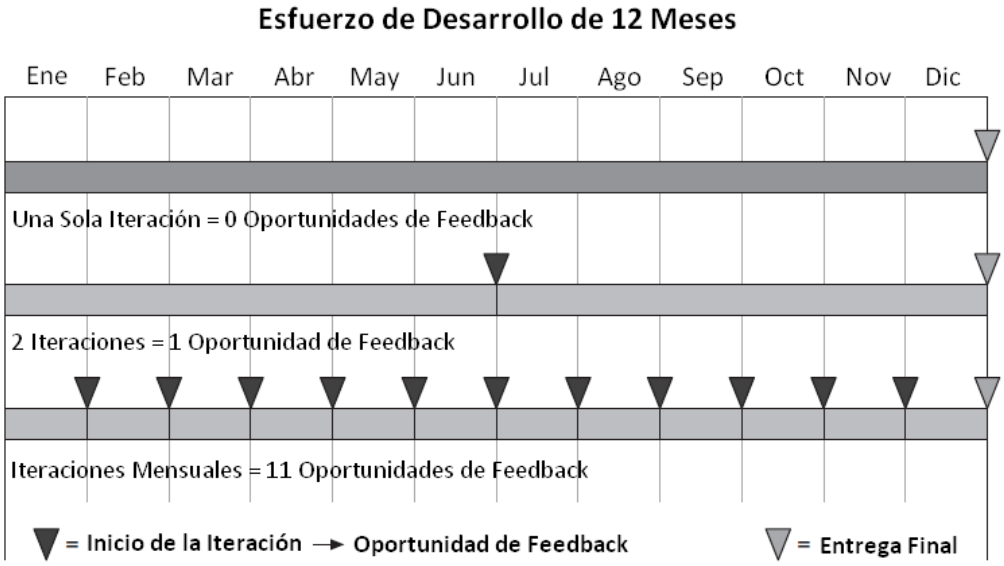


FIGURA 5.20 Iteraciones cortas aumentan las oportunidades de feedback

La retroalimentación de los clientes beneficia un esfuerzo de desarrollo de varias maneras. En primer lugar, suponiendo que se hizo una correcta utilización del concepto de requisitos priorizados, el feedback permite a los desarrolladores enfocarse en las características más importantes para el cliente. Cada iteración implementa un subconjunto de los requerimientos, comenzando con el más importante. Después de cada bucle la lista de requisitos es modificada en función del feedback del cliente; por lo tanto se añaden, eliminan y modifican requisitos, al mismo tiempo que se obtienen nuevas prioridades. La siguiente iteración nuevamente implementa un subconjunto de los requisitos actualizados en orden de prioridad, por lo tanto los desarrolladores siempre están implementando los requisitos más importantes para el cliente, incluso si las prioridades son completamente diferentes a las de las iteraciones anteriores.

El ciclo que se acaba de describir tiene un par de efectos secundarios importantes: los cambios en los requisitos se contabilizan de forma automática y el conocimiento del dominio adquirido en iteraciones iniciales se puede aplicar a los bucles posteriores, haciendo esas iteraciones más eficientes.

La retroalimentación frecuente limita la sobre inversión en un área del proyecto que se genera al darle al cliente la oportunidad de indicar cuándo una característica es “lo suficientemente buena”. A pesar de que un mayor refinamiento de la función puede ser posible y puede mejorar el producto final, el refinamiento continuo que se excede más allá de un cierto punto genera rendimientos decrecientes. El tiempo adicional gastado en características “suficientemente buenas” puede ser utilizado en las zonas donde el rendimiento es mayor.

Finalmente, la retroalimentación asegura que el producto adecuado sea desarrollado. Los requisitos iniciales se generan a menudo sin una buena comprensión de lo que realmente se necesita. Al proporcionar muchas versiones intermedias a través de iteraciones cortas, los desarrolladores permiten a los clientes obtener una mejor comprensión de los requisitos y ver el abanico de las posibilidades. El feedback de los clientes a intervalos regulares permite

a los desarrolladores cambiar la dirección del proceso, quitar o añadir funciones y detener el desarrollo en las zonas con rendimientos decrecientes.
El feedback también tiene un efecto positivo en la relación desarrollador-cliente. Un cliente que proporciona retroalimentación y ve que ésta se refleja en la siguiente iteración se sentirá parte del proceso. Además de fomentar la buena voluntad entre el cliente y el equipo de desarrollo, este sentimiento animará al cliente a dedicar más tiempo a la evaluación del producto y proporcionar una mejor respuesta.

5.8.5.3. CORREGIR EL RUMBO

Obtener retroalimentación de los clientes es la mitad del trabajo. Implementar en el producto la información obtenida es la otra mitad. Las iteraciones cortas permiten correcciones frecuentes, que se pueden utilizar para alcanzar el objetivo.
Hay dos tipos de problemas que pueden surgir durante el proceso de desarrollo: la falta de comunicación entre los clientes y desarrolladores, y los requisitos cambiantes. Se pueden abordar ambos problemas haciendo correcciones sobre la marcha en base al feedback del cliente.
La falta de comunicación entre los clientes y desarrolladores se manifiesta como características en el producto que no se comportan de la manera que pretende el cliente (Figura 12-7)

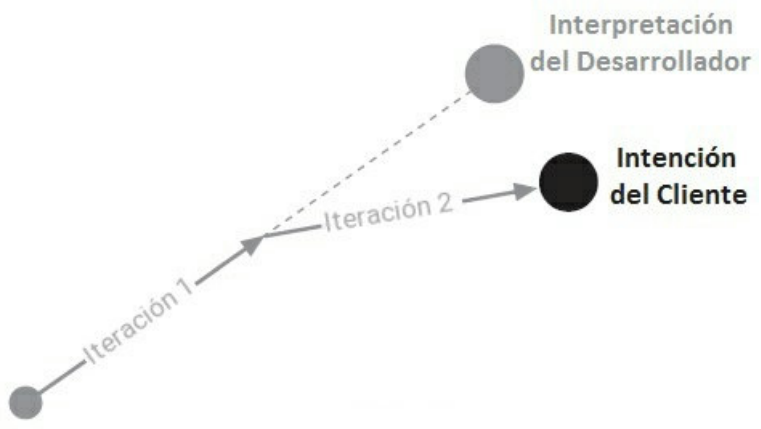


FIGURA 5.21 Corrección del rumbo debido a la falta de comunicación inicial

Los requisitos cambian cuando el cliente determina que los requerimientos originales ya no reflejan la funcionalidad requerida. Tales cambios pueden ocurrir por distintas razones, pero el resultado es una aplicación que ya no está en consonancia con las necesidades del cliente. Tal vez el cliente ha decidido cambiar de una aplicación cliente-servidor a una aplicación de servicios web que se ejecuta en un navegador. En este caso, a los desarrolladores se les da una nueva dirección y deben hacer una corrección de rumbo importante, como muestra la Figura 12-8.

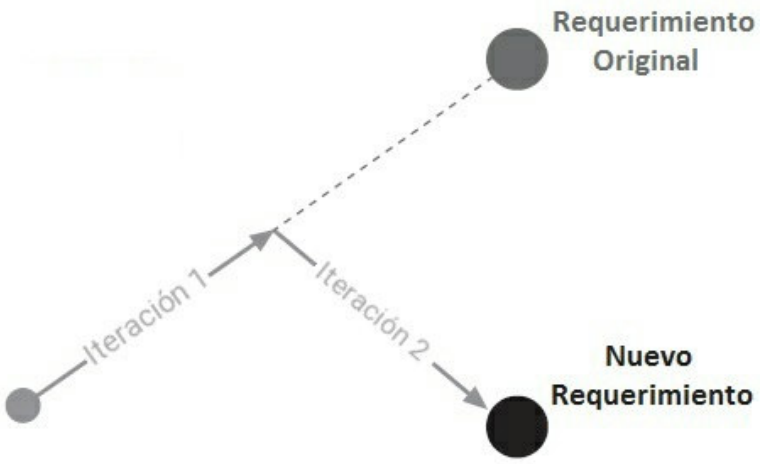


FIGURA 5.22 Corrección del rumbo debido a la evolución de las necesidades

La capacidad de hacer correcciones de rumbo depende de obtener retroalimentación por parte del cliente; mientras que hacer correcciones rápidas y eficientemente depende de las iteraciones cortas, las cuales generan la información necesaria en el momento oportuno.

5.8.5.4. DESARROLLANDO CON ITERACIONES CORTAS

Las técnicas que apoyan las iteraciones cortas son sencillas. Las dos primeras (requisitos priorizados e iteraciones de longitud fija) trabajan juntas para asegurar que los desarrolladores siempre estén trabajando para conseguir implementar los requisitos de mayor prioridad para el cliente. El resto de técnicas (mostrar el resultado de la iteración y entregar software funcional al cliente) ayudan a enfocar al equipo en el cumplimiento del objetivo de proporcionar valor y obtener una retroalimentación útil.

5.8.5.5. TRABAJAR CON REQUISITOS PRIORIZADOS

Los requisitos priorizados pueden ayudar a minimizar el tamaño del código base, pero también juegan un papel en las iteraciones cortas. Centrándose en los requisitos más importantes se ofrece la funcionalidad más útil al cliente, por lo tanto las iteraciones cortas que implementan las características más importantes en primer lugar ofrecen el mayor valor posible en el menor tiempo.

5.8.5.6. ESTABLECER LA LONGITUD DE UNA ITERACIÓN Y APEGARSE A ELLA

Aunque la mayoría de las metodologías ágiles recomiendan longitudes de iteración de dos a seis semanas, varios factores influyen en la misma, incluyendo la volatilidad del entorno, la experiencia con la metodología en uso, la capacidad técnica del equipo, su comprensión de las tecnologías usadas y la complejidad del propio sistema.
La elección de una longitud de iteración dentro del rango de dos a seis semanas, o en algunos casos fuera de ella, dependerá de cada uno de esos factores. Sin embargo, al

menos que la experiencia o una recomendación fundamentada digan lo contrario, el mejor punto de partida está en el medio de ese rango (cuatro semanas).
La comprensión del cliente sobre el dominio, el nivel de competencia y la inestabilidad del dominio en sí mismo se combinan para manejar la volatilidad del entorno. Por lo general, se encontrará una menor tasa de cambio cuando se desarrolla para un cliente que es la fuerza principal en un dominio y tiene muy poca competencia, que la que se encontrará en el desarrollo para una compañía que se lanza en un nuevo campo con muchos jugadores.
La experiencia del equipo también puede afectar la longitud de la iteración. Un equipo con mucha experiencia utilizando el proceso de desarrollo será más eficiente que uno con menos experiencia y será capaz de hacer más cosas en una iteración más corta. Lo mismo es cierto para la experiencia del equipo con las tecnologías que se utilizan. La necesidad de investigar una tecnología antes de usarla en un producto puede ralentizar el ritmo de desarrollo de manera significativa.
Finalmente, la complejidad del sistema puede afectar el ritmo de desarrollo. Los sistemas con muchos componentes externos suponen una carga de integración en el equipo y los sistemas que requieren una gran cantidad de desarrollo de la infraestructura pueden generar tareas que abarcan múltiples iteraciones.
Independientemente de la longitud de iteración elegida, la adhesión a la misma es muy importante, por lo que se debe establecer por adelantado una fecha final para cada iteración. Debe asegurarse que todos los involucrados (especialmente la gerencia) entiendan que la fecha no es negociable y que si el desarrollo comienza a retrasarse respecto de la planificación, se omitirá funcionalidad con el fin de cumplir con la fecha de lanzamiento. Una fecha de finalización es estricta permite dos cosas: asegurar que el cliente obtiene nueva funcionalidad para evaluar regularmente y mantiene enfocado al equipo de desarrollo.
El feedback de los clientes es uno de los principales impulsores de las iteraciones cortas; sin nueva funcionalidad para evaluar, los clientes no pueden proporcionar retroalimentación. Si un equipo de desarrollo permite que las fechas de finalización se dilaten, o peor aún, que postergar la fecha se convierta en un hábito, la capacidad de obtener e incorporar feedback estará en riesgo. Es mejor conseguir retroalimentación de menos funcionalidades que de ninguna en absoluto.
Las fechas de cierre de las iteraciones también enfocan al equipo en la tarea en cuestión. Un grupo que sabe cuándo tiene que entregar tiende a permanecer enfocado en los requerimientos, sobre todo cuando la iteración siempre termina con una demostración en presencia del cliente.

5.8.5.7. TERMINAR CADA ITERACIÓN CON UNA DEMOSTRACIÓN

Una demostración al final de la iteración sirve para dos propósitos: reconocer el esfuerzo del equipo de desarrollo y marcar oficialmente el final de la iteración.
Finalizar la iteración con una demostración da al equipo de desarrollo la oportunidad de mostrar lo que se ha venido haciendo a lo largo del bucle. Una reacción positiva por parte del cliente proporciona un sentido de realización y construye el entusiasmo para la siguiente iteración. Una demostración en el final de cada iteración también enfoca al equipo en que el trabajo se está realizando. Nadie quiere pararse frente al cliente y anunciar que ninguna nueva funcionalidad está lista, por lo que todo el equipo tiene un gran interés en implementar los requisitos y evitar tareas que no ayudan a alcanzar ese objetivo. También

mantiene presente el punto de que entregar nueva funcionalidad es lo suficientemente importante como para que finalizar algunos pocos requisitos sea mejor que terminar todos a medias.
Una demostración al finalizar la iteración es la declaración del equipo de que ha completado el trabajo que se acordó para la misma y significa la entrega oficial al usuario. Indica al cliente que el equipo de desarrollo está dedicado a alcanzar los hitos y cumplir las promesas.

5.8.5.8. ENTREGAR EL PRODUCTO DE LAS ITERACIONES AL CLIENTE

Aunque la demostración final de la iteración presenta a los clientes el producto de la misma, ésta no es suficiente para generar buen feedback. Los clientes necesitan tiempo para conocer el producto y aplicarlo a las tareas para las cuales eventualmente será usado. Sólo después de que los clientes han tenido tiempo de probar realmente la capacidad del producto podrán empezar a proporcionar información útil, por lo que el mismo debe ser entregado a los usuarios finales.
La entrega del producto puede significar algo tan simple como dar acceso a los clientes a un archivo de distribución en un servidor FTP. Los medios de entrega dependerán del tipo de producto en fase de desarrollo: las aplicaciones de clientes pueden distribuirse a través de CD/DVD, las aplicaciones web pueden ser distribuidas mediante la actualización de un servidor, las aplicaciones para móviles o iPads pueden requerir que se proporcione hardware específico para el nuevo software cargado. Sea cual sea el método, poner el software en las manos del cliente para que pueda utilizarlo en su propio entorno es esencial para obtener una buena retroalimentación.

5.8.5.9. LA FALACIA DEL DESARROLLO ITERATIVO

El desarrollo iterativo es un método eficaz para la creación de productos de software. Sin embargo, el deseo de crear planes deterministas y cronogramas a veces conduce a una implementación que marginaliza los beneficios del desarrollo iterativo. *La falacia del desarrollo iterativo* se basa en el siguiente par de suposiciones falsas:

El desarrollo iterativo es una serie de cascadas cortas.
El contenido de todas las iteraciones se debe definir en primer lugar dividiéndose los requisitos.

Estos supuestos conducen a un proceso, que se muestra en la Figura 12-9, en el que todos los requisitos son reunidos en el inicio del ciclo de desarrollo y una actividad de planificación por adelantado determina el contenido de todas las iteraciones repartiendo los requisitos en pedazos del tamaño de un bucle, por lo general con cada iteración centrada en un área diferente del producto. Por lo tanto el software se desarrolla a través de una serie de iteraciones, cada una de las cuales tiene las fases típicas del diseño, implementación y prueba en cascada. Aunque este enfoque es incremental (se añaden nuevas funciones en cada iteración), no incorpora la retroalimentación del cliente al reevaluar requisitos después de cada iteración. Las características implementadas durante la iteración se consideran completas y la que sigue simplemente se mueve a la siguiente función.

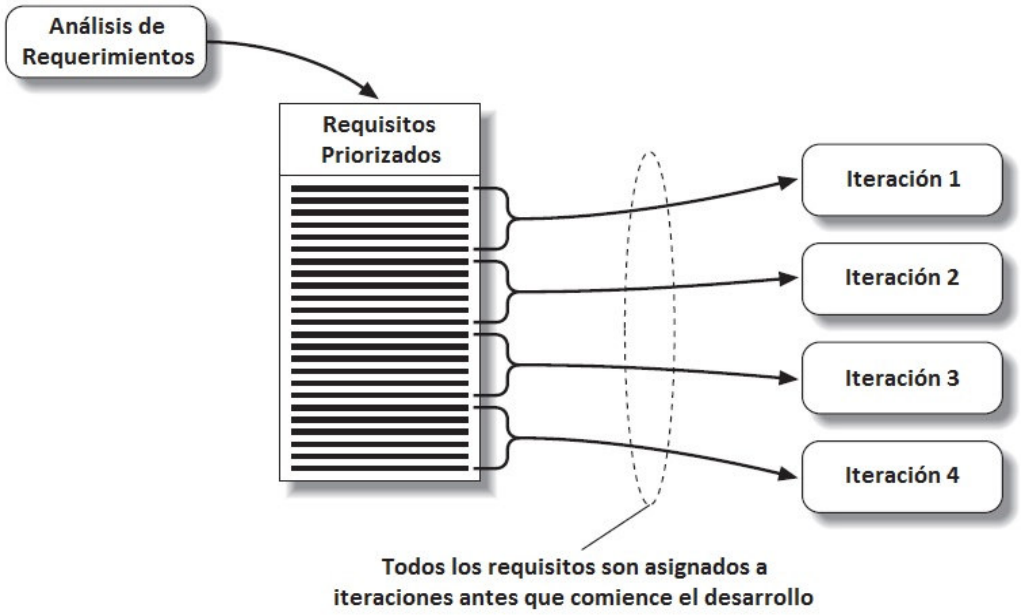


FIGURA 5.23 Requisitos asignados durante la planificación por adelantado

Dos problemas pueden surgir de este tipo de desarrollo iterativo: la entrada del cliente se ignora o se vuelven a planificar el esfuerzo en numerosas ocasiones. La planificación de todo el esfuerzo de desarrollo desde el principio no deja espacio para el cambio basado en el feedback del cliente. El equipo de desarrollo sigue adelante en base al plan, aun en frente de los comentarios de los clientes que indican que se dirige en la dirección equivocada. El resultado final de seguir con el plan original y hacer caso omiso de los comentarios de los clientes es que el equipo de desarrollo crea un producto equivocado.

La incorporación de la retroalimentación de los clientes después de cada iteración es importante, pero cuando se incorpora a través de la replanificación de las iteraciones restantes, el resultado es una gran cantidad de esfuerzo desperdiciado. Para un esfuerzo de desarrollo de 12 meses utilizando iteraciones de 1 mes, se planean 12 iteraciones al comienzo del proceso. Después de la primera iteración, las 11 restantes se vuelven a planificar en base a la retroalimentación del cliente. Después de la segunda iteración, las 10 restantes se vuelven a planificar y así sucesivamente hasta el final del desarrollo. Aunque la retroalimentación del cliente está siendo incorporada, gran parte del esfuerzo de planificación se desperdicia. La iteración final es planeada 12 veces, pero sólo se ejecuta el último plan.

La solución para ambos problemas es planificar solamente la siguiente iteración (corta) en detalle, como se muestra en la Figura 12-10. Las necesidades del cliente son capturadas por la lista de requisitos priorizados, por lo que la lista se actualiza al comienzo de cada iteración. La planificación de cada iteración es simplemente una cuestión de seleccionar un subconjunto de los requisitos de la parte superior de la lista.

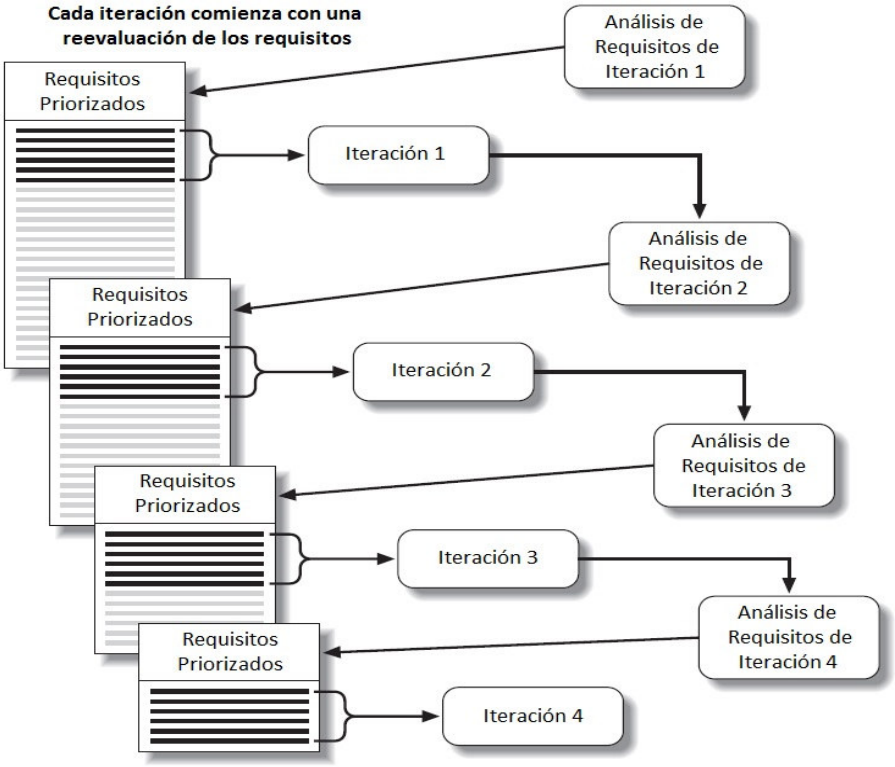


FIGURA 5.24 Requisitos reevaluados al comienzo de cada iteración

La planificación de iteraciones cortas de una en una a partir de una lista de requisitos actualizada garantiza que el equipo de desarrollo esté siempre trabajando sobre los temas que son más importantes para el cliente y que el producto final es lo que realmente necesita el cliente, sin desperdiciarningún esfuerzo tratando de planear demasiado lejos.

5.8.5.10. GRANDES TAREAS EN PEQUEÑAS PIEZAS

La mayor objeción al desarrollo en iteraciones cortas es que algunas tareas son demasiado grandes para ajustarse. La arquitectura del sistema a menudo se coloca en esta categoría, al igual que el desarrollo de una funcionalidad compleja que “no funciona al menos que todas las piezas estén en su lugar”. De manera superficial, estos pueden parecer argumentos válidos y en el desarrollo en cascada tradicional, lo son. Sin embargo, cambiar el paradigma de desarrollo se aborda ambos argumentos.
El diseño emergente permite que la arquitectura a nivel de componentes evolucione a medida que los requisitos son mejor definidos. Cuando una arquitectura evoluciona con el tiempo, las iteraciones cortas realmente ayudan al desarrollo. Los tiempos de respuesta rápidos que resultan de las iteraciones cortas proporcionan información oportuna sobre lo que funciona, permiten a los arquitectos tratar varios enfoques y posponer las decisiones importantes hasta que el sistema se entienda mejor.
El diseño emergente no exime a los arquitectos de los sistemas de toda la necesidad de pensar en el futuro. Por ejemplo, el intento de adaptar la seguridad en un sistema existente puede resultar en una gran cantidad de rediseño y reescritura. Sin embargo, el examen

adecuado de las cuestiones de seguridad desde el principio no requiere la implementación de un componente de seguridad completo en la primera iteración.
La implementación de una funcionalidad compleja a menudo no encaja en una iteración, lo que puede significar que el equipo de desarrollo no tiene ningún software funcional para liberar al final de la demostración. Sin embargo, una funcionalidad compleja normalmente se puede dividir en partes más pequeñas y menos complejas. La simulación de componentes inexistentes o funcionalidades incompletas mediante el uso de objetos de simulación permite al desarrollo continuar sin todas las piezas en su lugar. A medida que avanza el desarrollo, los componentes y funcionalidades se integran de forma incremental hasta que los objetos de simulación puedan ser sustituidos con el producto real. Este enfoque permite al equipo demostrar el software funcional después de cada iteración.

5.8.6. PARTICIPACION DEL CLIENTE

Si el objetivo del desarrollo de productos es crear algo que será utilizado por el cliente, es lógico pensar que su participación es una parte integral del proceso y su importancia nunca debe ser subestimada. Los conceptos de requisitos priorizados y correcciones de curso basadas en retroalimentación requieren de la participación activa del cliente.
Los clientes son la mejor fuente de información sobre el dominio del problema. Ellos conocen las tareas que se tienen que hacer, las condiciones en las que se debe llevar a cabo la solución y los objetivos que se están tratando de lograr con dicha solución. Rara vez tienen conocimientos sobre las tecnologías que se utilizan para implementar la solución.
Por el contrario, los desarrolladores saben las tecnologías. Ellos conocen las características de los últimos lenguajes y entornos de desarrollo, comprenden la relación entre las diferentes tecnologías y saben las maneras más eficaces para modelar problemas y soluciones. Lo que no conocen son las complejidades del dominio y el conocimiento de negocios que la solución debe incorporar.
Combinando el conocimiento de negocios de cliente con la experiencia técnica del equipo de desarrollo, se crea una dinámica de equipo que siempre genera las mejores soluciones. Los clientes proporcionan objetivos a los desarrolladores y estos muestran a los clientes el arte de lo posible, que a su vez permite a los clientes imaginar nuevas posibilidades. Ambos se inspiran mutuamente para lograr algo mejor de lo que podrían obtener individualmente.
La participación de los clientes es un camino de dos vías. Los equipos de desarrollo necesitan aportes de los clientes en forma de requisitos y prioridades, pero también deben mantener al cliente informado sobre el trabajo en curso. Los clientes necesitan seguir recibiendo retroalimentación y los equipos de desarrollo deben actuar en ella. La participación del cliente en todos los aspectos del proceso de desarrollo es la mejor manera de crear el flujo de información de dos vías.

5.8.6.1. INVOLUCRAR AL CLIENTE EN TODO EL PROCESO DE DESARROLLO

La participación tradicional del cliente consiste en la entrada durante la fase de análisis de requerimientos y la generación de solicitudes de cambios después de la entrega del producto final. El desarrollo de software Lean procura alcanzar un nivel mucho más alto de participación de los clientes, no solo participando en el desarrollo de los requisitos, sino también escribir, dar prioridad a las necesidades y definir las pruebas de aceptación

utilizadas para decidir si se han satisfecho los requisitos. A lo largo del ciclo de desarrollo, los clientes están a disposición de los desarrolladores para resolver las ambigüedades y responder preguntas. Con un sistema de integración continua que incluya auto despliegue, los clientes pueden comprobar el progreso de las pruebas del equipo y de usuario para garantizar que se cumplen los requisitos.

5.8.6.2. MANTENER INFORMADO AL CLIENTE

Dar participación a los clientes en el proceso de desarrollo significa mantenerlos informados de lo que está pasando. Se les deben comunicar los objetivos de cada iteración, así como el progreso del equipo hacia esos objetivos. Los clientes también deben ser informados de cualquier problema que surja durante el esfuerzo de desarrollo. Una aproximación inicial a los reportes de problemas aumenta la confianza entre el equipo de desarrollo y el cliente, y le da a éste la oportunidad de tomar ciertas medidas (como modificar la prioridad de los requisitos) para mitigar los retrasos causados por este tipo de problemas.
Proporcionar acceso al producto durante cada iteración puede ser una manera efectiva de mantener informado al cliente. El uso de la aplicación, incluso en un estado primitivo, permitirá ver lo que realmente está siendo desarrollado. Esto permite a los clientes ver sus comentarios incorporados al producto, lo que aumenta la confianza en el equipo de desarrollo.

5.8.6.3. ACTUAR SOBRE LOS COMENTARIOS DEL CLIENTE

Trabajar en función a la retroalimentación brindada por el cliente es la clave para mantener su participación a través de todo el esfuerzo de desarrollo. Un cliente que siente que sus recomendaciones están siendo escuchadas y puestas en uso probablemente continuará brindando esa entrada.
La retroalimentación generada mediante el uso de la salida de la iteración puede ser útil en la generación de nuevos requisitos, modificando o eliminando requisitos existentes. Sin embargo, la retroalimentación puede ser útil en cualquier momento durante el ciclo de desarrollo. Por ejemplo, los clientes que monitorean el progreso a través de una versión auto desplegable del producto pueden proporcionar orientación sobre los requisitos que actualmente se encuentran en fase de desarrollo.

5.8.6.4. TÉCNICAS

Cada esfuerzo de desarrollo es único. Diferentes clientes tienen diferentes formas de interactuar con el equipo de desarrollo y diferentes productos requieren distintos tipos de interacciones. Las siguientes técnicas proporcionan un punto de partida para desarrollar una relación de colaboración entre el cliente y el equipo.

Designar un Dueño del Producto: cada producto debe tener un campeón, comúnmente llamado dueño del producto, cuyo trabajo es representar las necesidades del cliente durante el ciclo de desarrollo. Él es la voz del cliente y es su trabajo definir el valor desde el punto de vista de su representado para el equipo de desarrollo. Es a menudo el responsable de mantener los requisitos priorizados y actúa como un punto de contacto común entre los desarrolladores y el cliente.

Involucrar al Cliente en la Redacción de Requisitos y Pruebas de Aceptación: comprometer a los clientes en la escritura de los requerimientos y pruebas ayuda a eliminar la confusión y malas interpretaciones cuando el equipo de desarrollo comienza a implementar los requisitos. En lugar de proporcionar a los desarrolladores una descripción del problema para ser descompuesta en requisitos, los clientes describen las características que desean y los criterios para determinar la finalización. Tener requerimientos escritos y pruebas también ofrece a los clientes la sensación de estar involucrados en el proceso y que se los mantendrá informados de lo que se está trabajando. Los requisitos definen lo que se debe implementar y las pruebas de aceptación demuestran que el requisito se ha cumplido. Cuanto más cerca se alinean estas dos cosas con las necesidades del cliente, más exitoso será el producto final al resolver los problemas para los que fue concebido, y la mejor manera de alienarlas es hacer que sea el mismo cliente quien las cree.

Informar el Estado del Proyecto: un aspecto para mantener informado al cliente es reportar el estado del proyecto. Los ítems de interés incluyen: contenido de la iteración actual, trabajo completado o pendiente en la iteración, resultados de las pruebas y problemas que afectan la culminación de la iteración.
Una manera de proporcionar acceso es a través de una aplicación de “tablero de control”. De fácil acceso por parte de los clientes, la gerencia y el equipo de desarrollo, los tableros de control aseguran que todos tengan una misma visión actualizada del estado del proyecto. Pueden incluir cosas tales como requisitos seleccionados para su aplicación durante una iteración, información sobre el trabajo restante en una iteración y resultados que indican que requisitos han pasado las pruebas de aceptación del cliente y se considera completa. Existen una serie de herramientas de gestión del ciclo de vida del software para Lean y demás metodologías agiles; la mayoría incluye una aplicación de tableros de mando y suele ser una aplicación web que proporciona fácil acceso para todos los participantes del proyecto.
Otra manera de brindar el estado detallado del proyecto que utilizan los equipos de desarrollo de software Lean es a través de reuniones diarias, donde los miembros del equipo discuten su progreso y los problemas actuales donde necesitan ayuda, permitiendo a los clientes escuchar todo lo planteado.

Proveer Acceso al Producto: al tener acceso durante la iteración, el cliente puede ver el progreso del trabajo. En lugar de esperar hasta el final para ver como se está implementando una característica particular, los clientes pueden ver la función tomar forma y proporcionar una retroalimentación inmediata.
La forma de acceso depende de la naturaleza del producto. Por ejemplo, las aplicaciones web pueden ser probadas a través de los servidores accesibles por el cliente y las aplicaciones de cliente pueden hacer uso de actualizaciones automáticas.

Crear un Camino de Retroalimentación: un mecanismo de retroalimentación fácil de usar es esencial para la recopilación de información de los clientes. Cualquier mecanismo que requiera demasiado esfuerzo no será utilizado y la calidad se verá afectada. El final del camino de realimentación debe ser fácil de usar para los desarrolladores, ya que cualquier tipo de información es inútil si nunca llega a ellos. Hay disponibles tres enfoques de realimentación para los equipos de desarrollo: reportes basados en formularios, aplicaciones

interactivas y reuniones cara a cara.
La retroalimentación basada en formularios, como el correo electrónico o las formas en las páginas web, es un buen método para utilizar con los clientes grandes e independientes. Aunque estos pueden sentirse menos conectados que con otros enfoques, éste permite un mayor volumen de información mucho mayor, ya que los desarrolladores no requieren interactuar con el cliente. También es apropiado para el seguimiento de los problemas individuales ya que cada envío de formularios puede ser numerado y registrado.

Las aplicaciones interactivas suelen ser aplicaciones web por su fácil acceso a través de los navegadores y la ventaja de no requerir que los clientes carguen software. Ejemplos de esto son los blogs y salas de chat. Cada uno de estos enfoques permite a los clientes interactuar directamente con los desarrolladores, dando inmediatez al bucle de retroalimentación. Las soluciones interactivas son eficaces cuando el número de clientes es pequeño y no pueden encontrarse con los desarrolladores. Las reuniones cara a cara son lo último en interacciones flexibles uno a uno y permiten a los clientes y desarrolladores trabajar juntos para resolver los problemas y aclarar ambigüedades. Sin embargo, para ser un mecanismo eficiente, las reuniones deben ser celebradas de forma regular, ser cortas y frecuentes, similares a los encuentros diarios entre los desarrolladores. Este enfoque puede resultar dificultoso con clientes que no tienen la posibilidad de reunirse y casi imposible con una gran base de clientes, pero es el mejor para los clientes con representantes in situ.
Encontrar Representantes *CRACK*: la participación del cliente se basa en tener representantes dedicados y ubicados. Barry Boehm y Richard Turner, en su libro *BalancingAgility and Discipline(Addison Wesley Professional, 2003)*, discuten la necesidad de tener representantes de clientes CRACK, que significa que son:

Colaborativos: trabajan bien con el equipo de desarrollo.
Representativos: comprenden el punto de vista del usuario final.
Autorizados: tienen el poder de tomar decisiones.
Comprometidos: comparten la dedicación del equipo de desarrollo para crear un buen producto.
Experto (*del inglés* ***Knowledgeable***): tienen el conocimiento y la experiencia necesarios para proporcionar orientación.

Los representantes CRACK proporcionan acceso inmediato al pensamiento de los clientes sobre los requisitos y pruebas de aceptación. Clarifican los requerimientos para los desarrolladores, validan las pruebas de aceptación y ayudan a mantener al equipo enfocado en las necesidades del cliente. Por otra parte, los representantes pueden formar parte del mecanismo de estado que mantiene informado al cliente y parte del bucle de retroalimentación que proporciona la entrada al equipo de desarrollo. También cumplen a menudo la función de dueños del producto, donde su profundo conocimiento de las necesidades del cliente les permite gestionar el manejo del desarrollo de los requerimientos.

En resumen, los clientes comprenden el dominio del negocio y los desarrolladores entienden las tecnologías. Combinar ambas cosas es una receta para el éxito.
La colaboración cliente-desarrollador requiere un flujo bidireccional de información entre ellos y se necesita que ambas partes participen activamente en la colaboración.

Los desarrolladores deben informar el estado del proceso y actuar en función de la retroalimentación, mientras que los clientes deben proporcionar requisitos priorizados, evaluar el producto y proporcionar información útil.
Se debe dar al cliente un papel activo en el desarrollo del producto y comprometerlos en la escritura de los requisitos y las pruebas de aceptación.
Mantener a los clientes informados e involucrados en función de los datos provistos en el estado del proyecto, darles acceso al producto a través del desarrollo y un mecanismo de retroalimentación fácil de usar.
Por último, un representante bien informado e involucrado tiene un valor incalculable en la creación y entrega de un producto que satisfaga las necesidades del cliente.

6. IMPLEMENTACIÓN DE LA METODOLOGIA

En esta sección se describe el diseño de la investigación junto con el razonamiento detrás este. Se puede decir que el diseño de una investigación está constituido principalmente por la filosofía que la fundamenta, el enfoque de investigación (estrategia de indagación), el método de investigación y el método de recolección y análisis de datos. Esta sección está estructurada de la misma manera, partiendo de un concepto más amplio (la filosofía de la investigación) hacía uno más específico (los métodos de recolección y análisis de datos).

6.1. FILOSOFIA DE LA INVESTIGACION

La filosofía de la investigación está relacionada con las afirmaciones de conocimiento realizadas dentro de un estudio, las cuales pueden definirse como suposiciones acerca de *qué* y *cómo* aprenderán los investigadores durante el proceso de investigación.
La elección de una filosofía de investigación depende principalmente del objetivo del proyecto. Como se ha descrito en la primera sección de este trabajo, el propósito de este estudio es mostrar el potencial que posee la metodología Lean para el desarrollo de aplicaciones, independientemente del ámbito de implementación. Con este objetivo en mente, podemos llegar a la conclusión de que la postura epistemológica predominante es positivista. Los estudios positivistas comienzan con la prueba o verificación de una teoría que es utilizada de una manera deductiva, ya que es encontrada en la teoría y la investigación es ideada para probarla.
El investigador positivista comienza con una teoría y luego recoge datos que apoyan la apoyan o la refutan. La teoría en la que se basa la investigación fue presentada en la sección del marco teórico.

6.2. ENFOQUE DE INVESTIGACION

El enfoque de la investigación tiene a seguir la filosofía de la misma. La elección del enfoque está fuertemente acoplada con el tipo de datos disponibles para el investigador. Sin embargo, en la investigación se presenta una particularidad. En la literatura, el positivismo se clasifica como un enfoque cuantitativo, pero en varios casos los datos recogidos fueron de naturaleza cualitativa. Esto se debe a que, por diversas limitaciones (sobre todo en lo que se refiere a experiencia en desarrollo de software) los investigadores no pudieron aplicar por si mismos la totalidad de los conceptos que fundamentan la metodología. Por lo tanto, los datos se basan en la experiencia propia del desarrollo del Sistema de Gestión Técnica de Fibra y en la recopilación de información aportada por el Ingeniero en Computación Gonzalo Haro y el Ingeniero Industrial José Ominetti, quienes brindaron una amplia e importante colaboración en la realización de este trabajo a través de su experiencia en la aplicación de muchos conceptos propios de la filosofía Lean. De esta manera, el estudio se basó en datos tanto cuantitativos como cualitativos, consistiendo estos últimos en descripciones (datos descriptivos) que tienen que ver con la generación de conocimiento y profundidad expresada en descripciones verbales.

6.3. METODO DE INVESTIGACION

La elección del método de investigación depende del objetivo del estudio. Como mencionamos anteriormente, el objetivo de este trabajo es mostrar el potencial que posee la metodología Lean para el desarrollo de aplicaciones, independientemente del ámbito de implementación. La teoría de validación suele estar relacionada con los experimentos o casos de estudio. En este trabajo se desarrollará una aplicación y será analizada como caso de estudio, determinando los distintos principios y prácticas aplicables en función del entorno específico de implementación y del tamaño y alcance del proyecto, ya que mucha de la teoría referida a Lean es fundamentalmente aplicable a grandes desarrollos emprendidos por equipos de trabajo con numerosos miembros; situación que excede los límites viables de este trabajo.
Se eligió el análisis de un propio caso de estudio para poder simular un entorno de desarrollo profesional y utilizar las técnicas recomendadas a fin de identificar indicadores acerca de la verdad o validez de la teoría Lean (mediante el análisis de los resultados y beneficios que se consideren obtenidos). Se pretende obtener comprensión sobre la aplicación de la filosofía Lean en las organizaciones de desarrollo de software y conocimiento profundo sobre los diferentes pasos y procedimientos que intervienen en los procesos de implementación de la metodología.

6.3.1. FASE TEORICA: RECOPILACION DE DATOS

Consideraremos que la investigación está constituida por dos grandes partes: la fase teórica y la fase práctica o empírica.
Durante la fase teórica, la investigación se centra en la recolección de información y obtención de conocimiento sobre la teoría Lean, sobre todo respecto a los principios y aspectos relacionados a los costos, calidad del software y respeto por las personas.
Los métodos utilizados para la obtención de información en esta fase se describen a continuación:

Revisión de literatura: durante la primera fase de este trabajo de investigación se lleva a cabo una revisión de libros relacionados, ponencias y artículos con el fin de obtener una comprensión general de la teoría Lean y sus principios de pensamiento, ayudando esto a definir el enfoque del estudio. Sobre la base del conocimiento existente en torno al desarrollo de software Lean, el foco de este trabajo está en validar la teoría Lean en términos de reducción de costos, mejora continua de la calidad y respeto por las personas. La lectura de los diversos libros, resúmenes y papers permiten realizar una colección de literatura relevante para el tema de este trabajo. Los datos recolectados permiten identificar y explorar los supuestos teóricos de la filosofía Lean, estableciendo las bases para el análisis de los datos.

Entrevistas: en esta fase se recogieron datos sobre la implementación de los principios y prácticas Lean a través de entrevistas, tomados en forma de audios o notas escritas. Se realizaron entrevistas a dos profesionales (uno de sistemas y el otro del rubro industrial) a fin de obtener información sobre la aplicación deherramientas afines a la filosofía Lean en cada uno de sus áreas específicas. En ambas entrevistas el foco de la conversación fue explorar

cuales fueron las características Lean (tanto en desarrollo de software como manufactura) que resultaron atractivas para sus respectivas organizaciones o actividades, que expectativas se tenían en términos de resultados y cuáles fueron los beneficios reales y los resultados obtenidos. Durante las entrevistas, las preguntas fueron de final abierto con el fin de facilitar la discusión del tema.

6.3.2. FASE PRÁCTICA: DESARROLLO E IMPLEMENTACION DE LA FILOSIFIA LEAN

Durante todo el tiempo que demandó la realización del presente trabajo, se fue ahondando en diferentes y múltiples fuentes de conocimiento respecto a la metodología de desarrollo Lean y se pudo ver que a través de los años muchos conceptos relacionados a la implementación de la filosofía Lean fueron emergiendo como una disciplina única relacionada con el movimiento Ágil, pero sin ser específicamente un subconjunto del mismo.
Definir de forma concisa la manera de aplicar Lean Software Development resultó para los realizadores de este trabajo un gran reto ya que no existe ningún método o proceso especifico que determine o reglamente su aplicación. Lean no es un metodología de trabajo equivalente a las que conocemos o hemos escuchado nombrar como por ejemplo Extreme Programming, Scrum o Desarrollo Basado en Pruebas.
En un principio, interpretamos que Lean Software Development debía ser considerado como uno de los muchos enfoques del movimiento Ágil respecto al desarrollo de software. De hecho, en muchas fuentes se pudo ver como se utilizaban los principios Lean para apoyar y fundamentar las razones por las cuales se consideraba que los métodos de desarrollo agiles eran mejores. Sin embargo, la metodología fue evolucionando a partir de la combinación de los principios y valores de la misma junto con ideas de desarrollo de productos del universo Ágil y conceptos procedentes del mundo de las metodologías tradicionales de desarrollo de sistemas.
En los últimos tiempos Lean ha aparecido como una fuerza que apunta al progreso del proceso de desarrollo de software, centrándose en mejorar el flujo de trabajo, la correcta administración del riesgo y la toma de decisiones. Se ha vuelto común y beneficioso el uso de herramientas como Kanban a la hora de administrar los proyectos Lean y se ha ido marcando una tendencia hacia el pensamiento de que en actividades como el desarrollo de software, la búsqueda de la mejora continua es más exitosa si uno se concentra en la administración del flujo en lugar de la eliminación de desperdicios.
Como dijimos anteriormente, buscar un método o pasos a seguir para la implementación de Lean fue una tarea decepcionante, por lo que se procedió a contemplar su implementación desde un enfoque inverso. Se comenzó a desarrollar el sistema y a medida que se fue avanzando con el trabajo, el mismo se fue ajustando para encuadrar dentro de los principios y valores básicos de la filosofía, tomando como fundamento la idea de que un proceso de ciclo de vida de desarrollo de software (sea ágil o no) o un proceso de administración de proyectos puede considerarse Lean si se cumple con los valores del movimiento Lean y los principios de Lean Software Development.
A continuación se detallarán los **valores, principios** y **procedimientos** que se buscaron implementar a lo largo del proceso de desarrollo que sirvió como caso de estudio y de esta manera plasmar el trabajo realizado en la aplicación de la filosofía Lean:

6.4. VALORES

Los valores son normas que nos permiten orientar nuestro comportamiento en función de ciertas creencias o principios. Nos ayudan a preferir, apreciar y elegir determinadas cosas en lugar de otras, o un comportamiento en lugar de otro. Lean Software Development no carece de los ellos y por el contrario, son el puntapié inicial en la búsqueda de su implementación.

Los valores sobre los cuales se trabajó fueron los siguientes:

1. ***Aceptar la condición humana.***
2. ***Aceptar que la complejidad y la incertidumbre son naturales para el trabajo de conocimiento.***
3. ***Trabajar con vistas hacia un mejor resultado económico.***
4. ***Trabajar siempre respetando el aspecto sociológico.***

A continuación se desarrollará cada uno de ellos:

1 - Aceptar la Condición Humana

El desarrollo de software es una actividad encasillada dentro de lo que denominamos "trabajo de conocimiento", que es el término que se refiere a toda actividad en la cual la principal labor es aplicar el saber ganado mediante el estudio o la experiencia. Las personas que realizamos este trabajo somos inherentemente complejas y muchas veces nos dejamos llevar por nuestras emociones e incluso rasgos animales a veces difíciles de controlar pese a considerarnos pensadores lógicos. Nuestra psicología debe tenerse en cuenta cuando se diseñan los sistemas o procesos en los que trabajamos, al igual que también debe incorporarse el comportamiento social, ya que somos por naturaleza emocionales, sociales, tribales y nuestro comportamiento cambia con el cansancio y la tensión. Los procesos correctos serán los que adapten e implementen la condición humana y no los que traten de denegarla o asumir un comportamiento lógico propio de las maquinas.
En lo referido al desarrollo objeto de estudio de este trabajo, se procedió a abordar este principio en dos pasos o etapas bien definidas:

- **Análisis de las tareas a realizar en su calidad de trabajo de conocimiento y sus diferencias con el trabajo presencial**: se realizó una comparación simple de ambos tipos de trabajo utilizando como modelo de referencia de trabajo presencial a los operarios del sector productivo de Coteca S.A. En el caso de estos, el factor principal es que el colaborador esté en su puesto de trabajo las horas convenidas. Por ende, cuantas más horas permanezca en la fábrica, mayor será su productividad medida en kilos de hilado diarios por persona. Dicho en términos simples, el mismo operario produce más hilo en un turno de doce horas que en uno de ocho horas.
  Sin embargo, fue muy evidente que durante el proceso de desarrollo de la aplicación para la administración del algodón no sucedía lo mismo que con los operarios. Estar sentados trabajando durante más horas no daba necesariamente mejores resultados y esto se debe a que el ejercicio que se debía hacer para realizar bien nuestro trabajo es muy diferente. Por supuesto que la cantidad de horas trabajadas tuvo su peso en términos de producción, pero también lo tuvieron el conocimiento y la inspiración.

Fue bajo esta situación de análisis en la que se hizo tangible que el éxito o el fracaso del proyecto dependía en gran parte de estar adecuadamente motivados y atacar los puntos necesarios para lograrlo.

- **Determinación de los puntos de motivación a reforzar y abordaje de los mismos**: una vez comprendida la importancia que tendría la motivación en nuestra tarea, analizamos y determinamos cinco puntos sobre los que deberíamos trabajar:

  I. Aprendizaje continuo: es el punto más importante y necesario en el desarrollo de un trabajo de investigación aplicado. Fue necesario que ambos participantes constantemente nos capacitáramos no solo la metodología de desarrollo Lean, sino también en todo lo referido al proceso de la hilandería de algodón (completamente desconocido para uno de los integrantes del equipo). Por otro lado también debimos capacitarnos constantemente en el lenguaje de programación elegido para la realización del proyecto, ya que ninguno de los integrantes se dedica a la programación en su trabajo habitual.

Tareas Realizadas
Revisión de teoría estadística
Revisión de teoría de organización industrial
Revisión de teoría de análisis y diseño de sistemas
Estudio de lenguaje de programación Visual Fox Pro
Estudio del proceso industrial hilandero
Estudio de técnicas de laboratorio textil
Revisión de estadísticas Uster
Estudio de manual Uster HVI
Estudio de procesos de logística

  II. Dirección y objetivos claros: durante el proceso de desarrollo, por momentos el grupo de trabajo se sintió sumido en un mundo completamente etéreo, y tener objetivos claros y establecer una dirección sólida permitió estar más enfocados en la persecución de los objetivos propuestos. Desde el punto de vista motivacional, los objetivos formales expresados en el trabajo no tuvieron tanto peso como los objetivos personales que representaban la realización del presente trabajo, siendo estos últimos verdaderos generadores de fortalezas ante las diversas dificultades encontradas.

  III. Flexibilidad de horarios: en lo referido a trabajo de conocimiento, la capacidad de concentración e inspiración fueron dos factores fundamentales, y es bien sabido que cuestiones personales y circunstancias cotidianas pueden afectarlos. Es por ello que se tomó la determinación de no manejarse por horarios estrictos al desarrollar el presente trabajo, aprovechando al máximo posible los momentos de fluidez de ideas y buen ambiente de trabajo. Sin embargo, es este punto en particular se debe tener cuidado, ya que es

muyfácil confundir el hecho de repartir los horarios a discreción con procrastinar.

Conductas Adoptadas
Establecer un mínimo de reuniones semanales para trabajar
Cambiar periódicamente el lugar de trabajo
Respetar las compromisos y situaciones personales
Exigencia mutua entre los integrantes del grupo

IV. Poder de decisión: todos los integrantes del equipo de trabajo debían tener el mismo poder de decisión sobre las tareas que se realizaban. Cuando una persona aprende, piensa y en función de ello produce algo, resulta frustrante no tener ningún tipo de participación en las decisiones sobre aspectos relacionados a los objetivos de se establecieron en el trabajo. Cualquier tipo de determinación respecto a acciones o criterios de abordaje del trabajo se tomaron estrictamente por consenso.

Etapas del Proceso de Toma de Decisiones Adoptado
Definición del objeto de decisión
Discusión superficial
Exposición de propuestas
Definición del criterio de selección de la propuesta
Evaluación de ventajas, desventajas y posibles obstáculos de implementación
Discusión profunda
Votación
Toma de decisión

V. Victorias y logros: en todo momento los integrantes del equipo reconocieron la importancia de valorar el trabajo bien hecho. Se adoptó la filosofía de felicitar verbalmente por un trabajo bien realizado e indicarse mutuamente los posibles errores cometidos, siempre de una manera positiva y proactiva para transmitir la sensación de apoyo en el crecimiento y no de crítica o penalización.

2 - Aceptar que la complejidad y la incertidumbre son naturales para el trabajo de conocimiento

Como ya es sabido, el comportamiento de los clientes y el mercado son imprevisibles. También pueden serlo el flujo de trabajo a lo largo de un proceso y el comportamiento de un conjunto de personas trabajando. Por último, los defectos y el trabajo por duplicado pueden aparecer en cualquier momento. En función de todo esto, el desarrollo de software conlleva riesgos o comportamientos aparentemente aleatorios en muchos niveles y el propósito, los objetivos y el ámbito de los proyectos suelen cambiar mientras se entrega el producto.

Parte de esta incertidumbre y variabilidad se pueden conocer a través de su estudio y cuantificación, aunque inicialmente sean desconocidas, así como administrar sus riesgos. Pese a esto, cierta variabilidad es desconocida de antemano y no se puede anticipar de forma adecuada. Como resultado, los sistemas desarrollados bajo la filosofía Lean deben poder responder a los eventos inesperados y adaptarse a las circunstancias fluctuantes. Por lo tanto, todo proceso que se ajuste a Lean Software Development debe existir dentro de un marco que permita la adaptación a cualquier contingencia.
Durante el desarrollo del trabajo, la necesidad de adoptar este valor se hizo tangible a medida que se fue profundizando el estudio de la metodología Lean Software Development. Al principio del desarrollo del proyecto se tomó como principal fundamento de aplicación la eliminación de desperdicios para luego ir migrando hacia la optimización del flujo de trabajo. Este cambio de foco en el proceso de desarrollo fue beneficioso desde el punto de vista del proceso de desarrollo, pero obligó a los integrantes de equipo de trabajo a flexibilizar su postura y adaptabilidad a los cambios. Varios conceptos y procedimientos aplicados se fueron implementando a base de prueba y error y muchas veces esto produjo pérdidas de tiempo y la necesidad de rehacer trabajo (ambos principales ejemplos de desperdicio), pero esto ayudo a aprender a identificar los factores generadores de desechos y a su prevención en pasos siguientes del desarrollo.

3 - Trabajar con vistas hacia un mejor resultado económico

Todas las actividades comprendidas en un proyecto de Lean Software Development deben estar centradas en generar un mejor resultado económico. Desde un punto de vista general, el capitalismo es aceptable cuando contribuye al valor empresarial y beneficia al cliente. Expuesto de una manera más detallada, los inversores y los propietarios de los negocios merecen una mayor rentabilidad de la inversión; los empleados y los trabajadores merecen una tarifa justa por su esfuerzo de trabajo y los clientes merecen un buen producto o servicio que ofrezca las mejoras y ventajas prometidas a cambio de un precio justo. En función de estos conceptos, los mejores resultados económicos implican una entrega de más valor al cliente a menor precio mientras se administra el capital invertido de la mejor manera posible.
La idea de obtener un mejor resultado económico estuvo plasmada desde el principio del proyecto. Como cualquier empresa, Coteca S.A. busca obtener el mayor rédito posible dentro de un contexto económico que dista de ser el más favorable, debiendo constantemente optimizar recursos y reducir costos para mantener su posición de líder en un mercado agresivo y sumido en constantes vaivenes económicos.
Dentro de los costos fijos que tenemos en cualquier empresa textil, la mano de obra y la materia prima son los principales, y una eficiente administración de esta última ayuda a reducir drásticamente otro factor crítico: los costos de defectos y reclamos de clientes.
Por motivos de confidencialidad no se pudo tener acceso a las cifras reales que maneja la empresa, pero conceptualmente es claro que cualquier reducción de los reclamos de clientes y los defectos del hilado se traducen en mejores ganancias económicas para la empresa. Con la implementación del Sistema de Gestión de Fibra se pretende reducir casi por completo la ocurrencia de reclamos por barrado en blanco y en teñido y brindar mayor estabilidad y previsibilidad a las características físicas del hilado en función de las variaciones de las campañas algodoneras y calidad de lotes de proveedores.

4 - Trabajar siempre respetando el aspecto sociológico

Este concepto está estrechamente relacionado con el punto anterior. No se deben obtener mejores resultados económicos a expensas de las personas que están realizando el trabajo. En las empresas lamentablemente es una situación que se ve con más frecuencia de lo que uno desearía, ya que la mano de obra siempre fue la variable de ajuste por excelencia. Es por esto que Lean aporta un enfoque concentrado en respetar y valorar a las personas mediante la aceptación de la condición humana y aboga por proporcionar sistemas de trabajo que contemplen la naturaleza psicológica y sociológica de las personas. Crear un buen sitio para hacer un gran trabajo es un valor básico en Lean Software Development y en todo momento se intentó constituirlo durante el desarrollo de este trabajo, fomentando un clima de respeto y cordialidad constantemente, estando atentos a estados de ánimo y situaciones personales del otro, buscando consolidar la sensación de tener compañeros de trabajo preocupados por el proceso de desarrollo y por las personas que los rodean.

6.5. PRINCIPIOS Y PROCEDIMIENTOS

6.5.1. Análisis del proyecto y determinación de la cadena de flujo de valor

Como primera instancia en el proceso de desarrollo de la aplicación objeto del presente trabajo, se precedió a realizar un primer análisis general para determinar las características básicas del cliente al cual apunta la aplicación. Para ello se formularon cinco simples preguntas que darán un acercamiento conciso para continuar con el siguiente paso. Las interrogantes fueron:

- ***Primera pregunta: ¿Quién es nuestro cliente?***
  Respuesta: *nuestro cliente es COTECA S.A., una empresa hilandera ubicada en el parque industrial El Pantanillo. Produce hilados para tejeduría de 100% algodón y tiene una producción mensual promedio de 350 toneladas y un consumo de fibra virgen de algodón de aproximadamente 400 toneladas. El total de la producción se comercializa exclusivamente en el mercado nacional para la realización de tejidos planos (denim y gabardina) y de punto (frisa y jersey), que luego son utilizados para la confección de prendas de vestir. La fábrica pertenece a la firma TN&PLATEX S.A.; empresa líder en el rubro textil argentino y con una estructura conformada por seis plantas distribuidas en cuatro provincias y oficinas centrales en Buenos Aires.*

- ***Segunda pregunta: ¿Quién será el usuario?***
  Respuesta: *el usuario final de la aplicación en desarrollo será en Encargado de Materia Prima de la fábrica. El mismo tiene la tarea de administrar el algodón en bruto desde el punto de vista de la logística y desde la calidad. Esta persona realiza las mezclas de algodón que luego son ingresadas a la línea de producción para su consumo. Las mezclas de materia prima (denominadas "calles") son estructuras de fardos de algodón que en su totalidad suman entre 12 y 14 toneladas que son ingresadas diariamente para alimentar el proceso hilandero. Las calles deben ser armadas respetando estrictos parámetros de calidad propios del algodón y cuyo manejo ineficiente es responsable del 80% de los reclamos recibidos por los clientes con las significativas pérdidas económicas que ello representa.*

- ***Tercera pregunta*: ¿Quiénes son las partes interesadas?**
  Respuesta: las partes interesadas evidentemente son los mandos gerenciales y mandos medios representados por el Encargado de Materia Prima. En primer lugar, la gerencia tiene interés en la implementación de un nuevo sistema de manejo y administración de la materia prima ya que consideran que es necesario un cambio de paradigma respecto al criterio utilizado para el armado de calles que provea mayor estabilidad y previsibilidad en las mezclas de algodón. Esto repercute de manera directa en la satisfacción del cliente y por ende en la cantidad de reclamos a raíz de defectos en la tela producidos por el manejo ineficiente de la fibra del hilado. En segundo lugar, el Encargado de Materia Prima tiene fuerte interés en el desarrollo de una herramienta que le permita un manejo más sencillo y eficiente de la fibra de algodón utilizada en el armado de calles. Esto le permitirá mejorar su desempeño y evitar en la mayor manera posible los reclamos de clientes atribuibles a su trabajo. La gerencia posee interés comercial y monetario mientras que el mando medio considera necesaria la herramienta desde un punto de vista técnico y motivacional.

- ***Cuarta pregunta*: ¿Cómo se transmiten las necesidades del cliente?**
  Respuesta: en el caso puntual de este proyecto de desarrollo, el cliente confía la determinación de las necesidades al futuro usuario del sistema y el mismo es uno de los integrantes del equipo de desarrollo. Por lo tanto las necesidades son completamente conocidas y compartidas a los otros integrantes de equipo de manera oral y a través de documentación poco formal pero altamente didáctica y práctica (bosquejos y diagramas A3).

Luego de estudiadas y determinadas las respuestas anteriores se procede a planear la cadena de flujo de valor de todo el proceso de desarrollo, separando el proyecto en los bloques fundamentales de tareas para ir abordándolos uno por uno e intentando respetar los tiempos estimados que se expresan en el diagrama.
Para la confección del mismo se investigó sobre herramientas específicas para la tarea, pero todas las aplicaciones encontradas requerían el pago de licencia e invertir tiempo en el aprendizaje de su uso. Como ejemplo de las mismas podemos nombrar a *Microsoft Visio* y *Edraw Max.* También se probó la herramienta web *Creately* (https://creately.com/value-stream-mapping-tool) pero también requería un pago monetario para su uso.

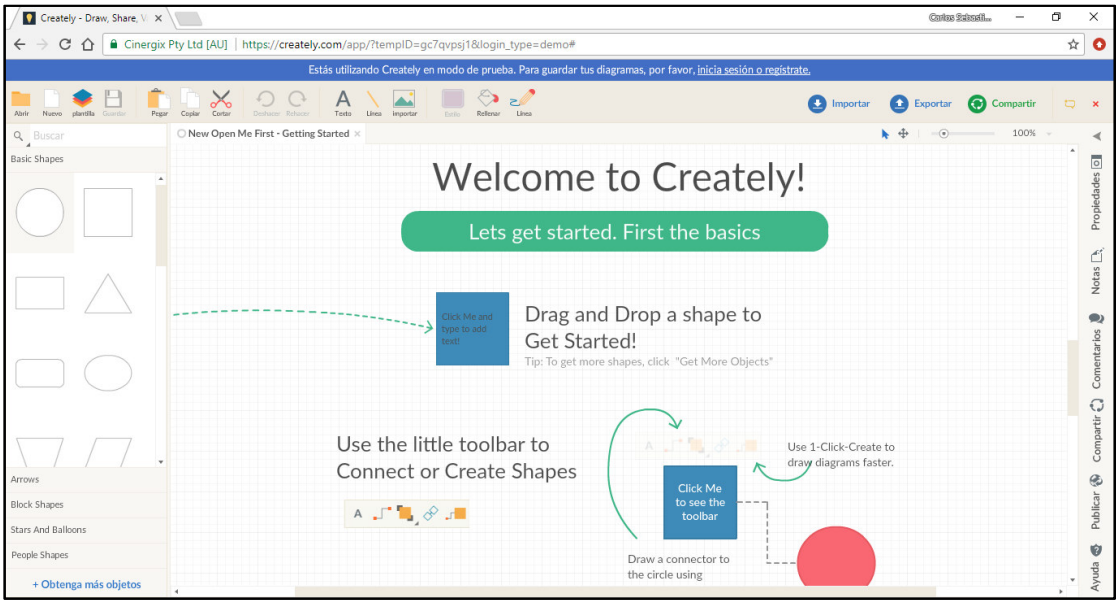


Figura 6.1 Herramienta web Creately

La filosofía Lean profesa la reducción de costos y tiempos; por lo tanto se consideró apropiada la búsqueda de una herramienta sencilla de aplicar y sin ningún costo. Por lo tanto se llegó a la conclusión de que el diagrama VSM (*Value String Map*) podía realizarse fácilmente con Microsoft Excel, obteniendo un resultado rápido, eficaz y sin ningún costo adicional.

A continuación se aprecia el diagrama del VSM realizado en función de cuatro iteraciones bien definidas que representan los módulos más importantes del sistema a desarrollar. En cada iteración se especifican las tareas que intervienen y que son críticas para la culminación de esa etapa y el avance del proyecto. La cadena de flujo de valor (o mapa de flujo de valor, dependiendo del autor) demostró ser una útil herramienta para ilustrar, analizar y mejorar los pasos necesarios para entregar el producto al cliente.

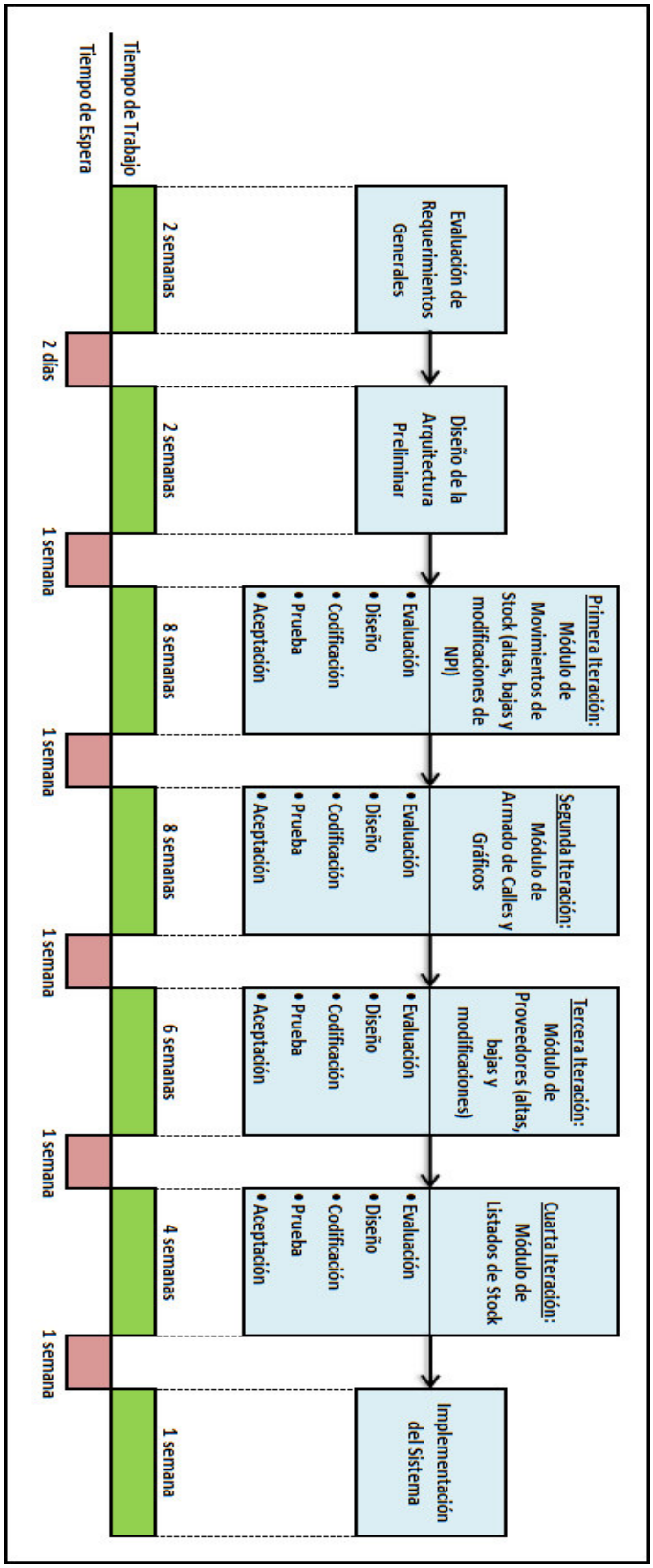


Figura 6.2 Mapa de flujo de valor

6.5.2. Identificación y reducción de desperdicios

En función de lo plasmado en la cadena de flujo de valor se inició con el proceso de desarrollo, abordando cada iteración con un enfoque basado en reconocer y eliminar las tareas o elementos que pudiéramos clasificar como desperdicio. Se reconocieron los siguientes:

- Conocimiento Deficiente y Reaprendizaje
- Documentación Innecesaria
- Características Extra
- Esperas y Cambios de Tareas

CONCEPTO		
Conocimiento Deficiente y Reaprendizaje		
ELEMENTOS IDENTIFICADOS	**JUSTIFICACION Y MEDIDAS TOMADAS**	**RESULTADO OBTENIDO**
Planificación deficiente y tareas en paralelo.	Se identificó como una fuente potencial de desperdicio. Se tomó en consideración el conocimiento previo al momento de realizar la asignación de tareas para evitar que el trabajo comenzado por un integrante sea completado por otro.	Se eliminó el costo de tiempo invertido en reaprendizaje. Cada persona trabaja en lo que es idóneo y completa esa tarea antes de seguir con la siguiente.
Comunicación y gestión inapropiada de la información.	El conocimiento e información formal es de fácil transmisión al encontrárselo en formato escrito. Sin embargo la mayoría del conocimiento compartido entre los integrantes del proyecto fue de naturaleza tácita, obtenido en su mayor parte a través de la experiencia. Para facilitar su transmisión se hizo uso de hojas que se adaptaran al formato del Informe A3 de Toyota. Esta planilla probó ser muy útil como soporte en la resolución de problemas y toma de decisiones.	Los formularios A3 mostraron tener una utilidad al permitir tener acceso rápido y fácil de interpretar a problemas planteados y sus respectivas propuestas para solucionarlos. Más adelante se muestra su utilización durante el desarrollo del proyecto.

Tabla 6.1

CONCEPTO		
Documentación Innecesaria		
ELEMENTOS IDENTIFICADOS	**JUSTIFICACION**	**RESULTADO OBTENIDO**
Especificación de requisitos escrita	En el caso de este desarrollo en particular, cliente y desarrolladores cumplen el mismo rol. El desarrollador tiene perfectamente claro cuáles son las necesidades y objetivos del producto final a raíz de llevar años trabajando en el tema. Se reemplazó cualquier tipo de documentación formal por una comunicación fluida entre los integrantes del equipo, diagramas a mano alzada y en formato A3 de Toyota que sirvieron para graficar algunos conceptos transmitidos.	Muy poco tiempo de planificación y escasos recursos utilizados en la confección de documentos escritos
Entrevistas escritas con clientes	Al igual que en el punto anterior, no se las consideró necesarias desde el punto de vista del proceso de desarrollo de la aplicación. Toda la información sobre el proceso productivo y necesidades que justifiquen la implementación son de vasto conocimiento por parte del equipo de trabajo. Por lo tanto se las consideró innecesarias.	Muy poco tiempo de planificación y escasos recursos utilizados en la confección de documentos escritos
Manual de Usuario	El manual de usuario es un documento muy importante en cualquier proyecto de desarrollo de software, pero las características particulares de la aplicación en cuestión y la relación dada entre cliente y desarrollador hacen que el mismo se haya considerado innecesario. El sistema tendrá únicamente un solo usuario y en caso de cambiar, las políticas de la empresa requieren estrictamente un proceso de capacitación persona a persona, trabajando conjuntamente hasta hacer el cambio definitivo de tareas. Un manual de usuario en la práctica nunca es utilizado.	Muy poco tiempo de planificación y escasos recursos utilizados en la confección de documentos escritos

Tabla 6.2

CONCEPTO		
Características Extra		
ELEMENTOS IDENTIFICADOS	**JUSTIFICACION Y MEDIDAS TOMADAS**	**RESULTADO OBTENIDO**
Interfaz de usuario	El primer punto en el cual se determinó que se generaba desperdicio fue en la interfaz de usuario. Se optó por un diseño simple y visualmente austero, pero que cumpliera todas las necesidades desde el punto de vista de la funcionalidad. En el caso de este desarrollo en particular, el usuario final da muy poca importancia a "como se ve" el entorno y vuelca todas sus exigencias hacia los resultados numéricos que son críticos en el proceso productivo algodonero.	Eliminando requerimientos estéticos se obtuvo un sistema simple, fácil de desarrollar y usar, y al mismo tiempo completamente funcional.
Control de Inicio de sesión	Esta característica se considera común en cualquier sistema que pueda tener múltiples usuarios. El caso del trabajo desarrollado, esta funcionalidad fue innecesaria y por lo tanto considerada desperdicio. El sistema tendrá un único usuario y el mismo estará instalado en una computadora personal brindada por la empresa. Nadie más en toda la fábrica tendrá autorización para hacer uso de esa computadora y sistema. Es por esto que una función tan común fue considerada desperdicio de recursos y tiempo en este sistema.	La seguridad de la información es brindada por el inicio de sesión del sistema operativo al encender la computadora o reactivarla luego de una suspensión. El usuario valoró la supresión de un paso de control que no consideraba necesario.
Listado de stock fardo por fardo.	Es otra función considerada habitual y útil pero desestimada en este sistema. Como mencionó anteriormente, se planteó un nuevo formato de archivo de texto con la información de los NPI. El listado contenido en cada plano pasó a tener información de promedios del camión y no valores puntuales fardo por fardo. Esto hizo que se volviera evidente que lanzar un listado completo del stock fardo por fardo seria poco útil ya que habría mucha información numérica repetida (un camión completo en este listado estaría representado por más de 100 registros exactamente iguales).	Se reemplazó el listado fardo por fardo del stock por uno nuevo que contiene el stock completo expresado en camiones con su cantidad de fardos y valores promedios de calidad. Este reporte resultó ser mucho más simple de interpretar y de utilidad ampliamente superior.

Tabla 6.3

CONCEPTO		
Esperas y Cambios de Tareas		
ELEMENTOS IDENTIFICADOS	**JUSTIFICACION Y MEDIDAS TOMADAS**	**RESULTADO OBTENIDO**
Esperas por prueba y aprobación del cliente	La filosofía Lean tiene como una de sus premisas hacer las cosas bien “a la primera” para aumentar la posibilidad de satisfacer al cliente lo más rápido posible. En el caso del desarrollo de este trabajo, jugó un papel trascendental el rol de desarrollador y cliente de los integrantes del equipo. Es por esto que las necesidades del cliente estuvieron presentes en todo momento guiando el rumbo del proceso. Al final de cada iteración se tenía completa certeza de que el resultado era el esperado y por lo tanto el tiempo formal de prueba de clientes quedaba suprimido.	Esta condición particular de cliente-desarrollador hizo que los tiempos de prueba y aprobación de la aplicación se vieran considerablemente reducidos. También se eliminó el tiempo necesario para introducir al cliente en el entorno y uso de la aplicación. En el proyecto se consideró estos tiempos como desperdicios.
Cambios de tareas	Los cambios de tareas se consideraron desperdicios por el costo de tiempo de reaprendizaje necesario para que el nuevo encargado de la realización se interiorizara en el trabajo. En cambio, se decidió que ante cualquier cuello de botella que apareciera se atacaría al mismo con la ayuda de los dos integrantes. Normalmente los problemas presentados tuvieron su origen en desconocimiento técnico del lenguaje de programación o sobre el proceso textil.	Se consolidó la metodología de atacar cuellos de botella de a dos. De esta manera se conjugaron los conocimientos de todo el equipo para sortear los problemas encontrados, aportando cada integrante los conocimientos sobre los cuales tenía más experiencia y estudio.

Tabla 6.4

6.5.3. Utilización de tablero Kanban

Como paso siguiente al proceso de desarrollo se procedió a la implementación de un tablero kanban en donde se representaron las tareas a realizar a través de tarjetas de colores con un proceso específico en cada una de ellas. Las tareas se determinaron en función del criterio de los integrantes del equipo y se dividieron en grupos tomando como modelo orientativo el mapa de flujo de valor.
En primera instancia se abordó la etapa de determinación de requisitos y diseño preliminar del sistema. Se consensuaron las tareas que se fueron identificando como críticas para el avance del proyecto y anotándolas en tarjetas que se dispusieron sobre una pizarra dividida en el formato tradicional kanban (tareas pendientes, tareas en proceso y tareas finalizadas). Cada integrante seleccionó una tarjeta para comenzar y las mismas migraron a la columna de tareas en proceso.
Una vez finalizada la tarea abordada, se colocó la tarjeta en la columna de tareas finalizadas y se procedió a seleccionar una nueva tarjeta de la columna de pendientes.

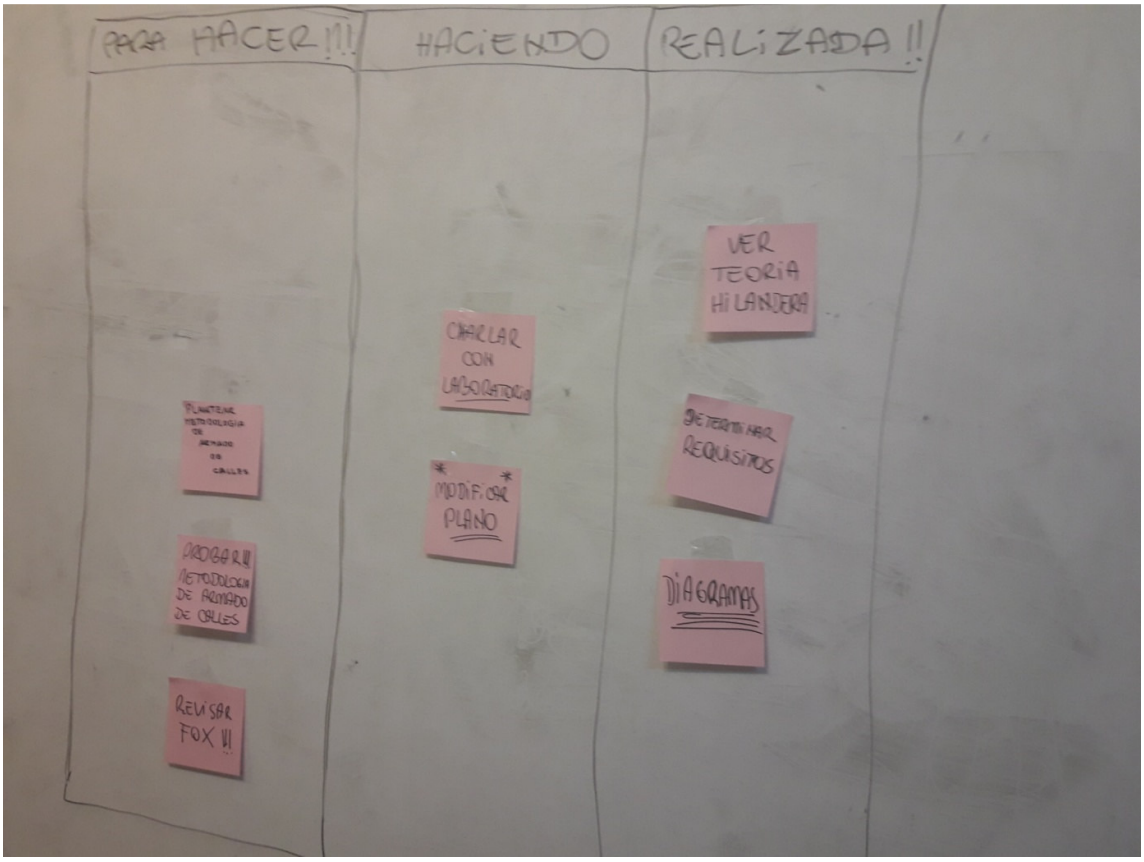


Figura 6.3 Tablero Kanban de Primera Etapa

Se siguió aplicando el mismo criterio hasta finalizar las tareas pendientes. Luego se comenzó nuevamente el proceso abordando el segundo módulo del desarrollo de la

aplicación, comprendido por las tareas relacionadas a las altas, bajas y modificaciones del stock de fibra y desperdicios, armado de calles y generación de gráficos.

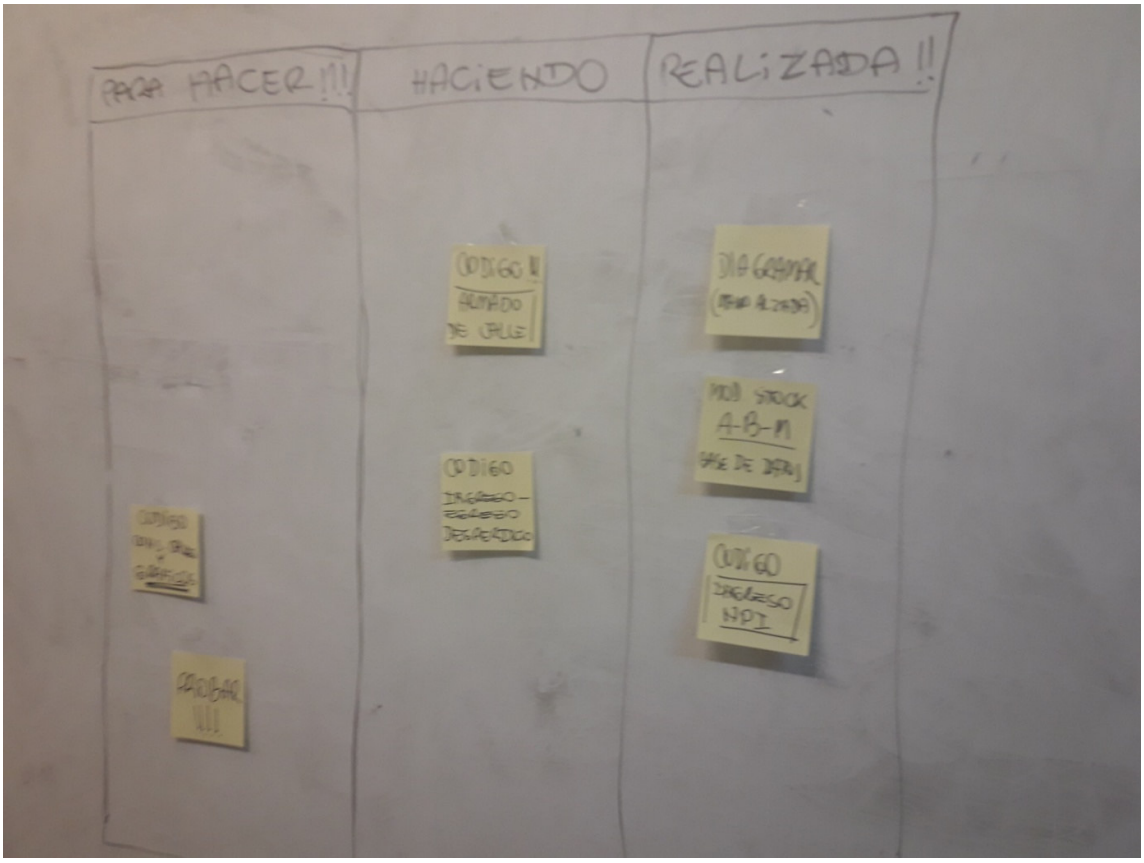


Figura 6.4 Tablero Kanban de Segunda Etapa

Luego de finalizada esta etapa, se realizó nuevamente el proceso para los módulos de altas, bajas y modificaciones de proveedores y listados en general. Para finalizar se incluyó la tarea puntual de implementación como parte del tablero kanban.

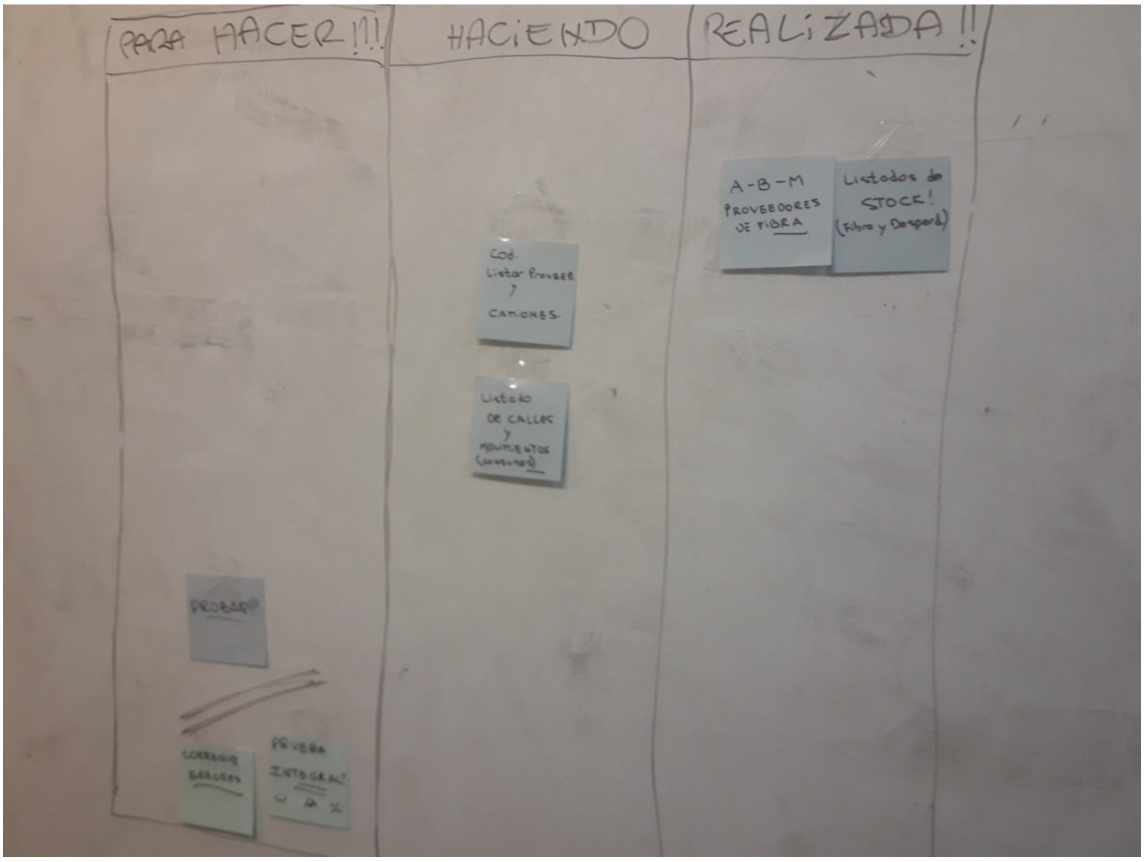


Figura 6.5 Tablero Kanban de Tercera Etapa

La utilización de esta herramienta aportó un gran componente de organización al proceso, permitiendo tener a la vista el estado del conjunto de tareas completo y de una manera sencilla de interpretar. El principio fundamental de su utilización fue realizar cada tarea de forma correcta la primera vez, lo cual no permite minimizar tiempo pero ayuda a garantizar calidad en el trabajo realizado, lo cual se ajusta a la filosofía aplicada en la metodología de trabajo.

Otra ventaja que brindó el uso de kanban fue la posibilidad de realizar un constante monitoreo del flujo de trabajo. Las tareas escritas en las tarjetas fueron ordenadas en función del mapa de flujo de valor, por lo tanto se pudo tener a la vista constantemente la manera en la que transcurría el proceso.

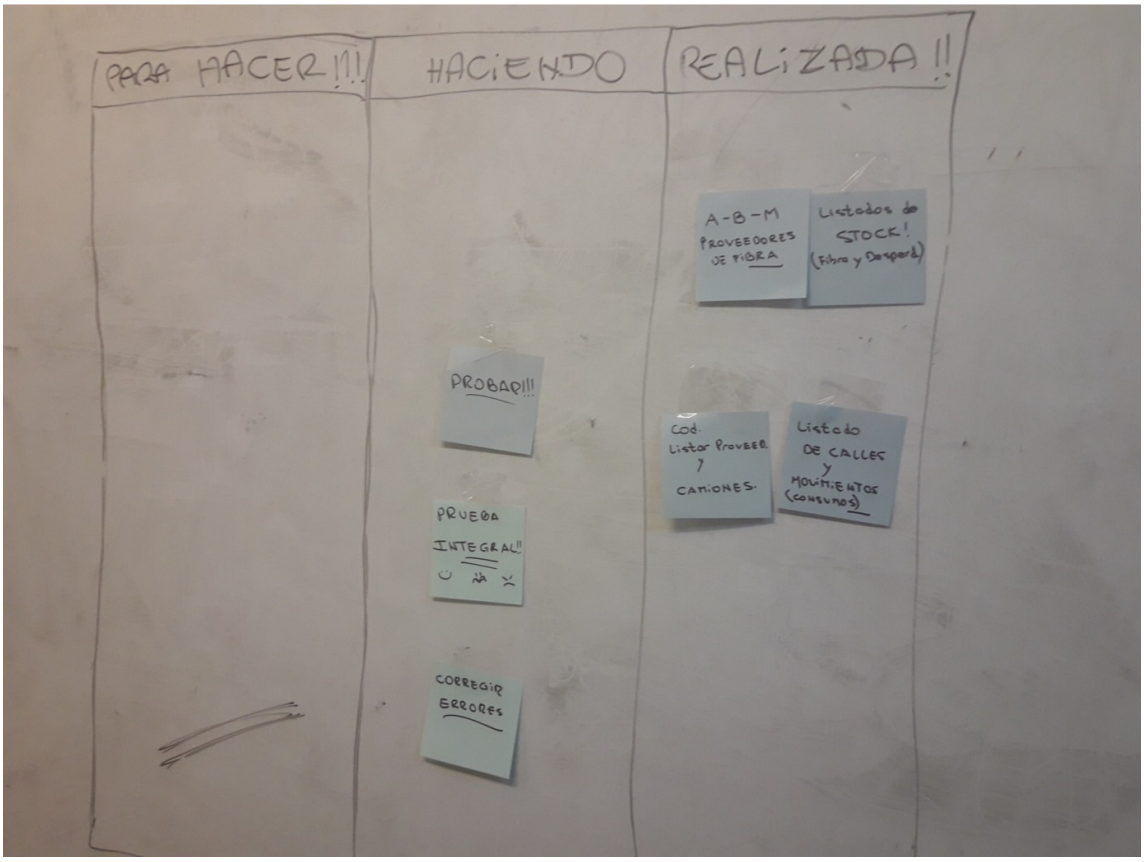


Figura 6.6 Tablero Kanban al Finalizar el Proceso

6.5.4. Asumir el rol de cliente y obtener retroalimentación

En las grandes organizaciones el desafío del desarrollo de software tiene una gran predisposición de aplicar un proceso lo más disciplinado posible, cumpliendo estrictamente con los requisitos documentados de forma completa, donde se debe conseguir que todos los contratos con el cliente sean por escrito, los cambios que se realicen sean examinados muy cuidadosamente y cada requisito y especificación debe traducirse a código. Todo esto se corresponde al compromiso de hacer controles adicionales en un entorno dinámico, dilatando el ciclo de retroalimentación. En la mayoría de los casos el aumento de la retroalimentación, y no su disminución, es una manera más eficaz de hacer frente a grandes proyectos de desarrollo y entornos problemáticos.
Siempre se dijo que la participación del cliente es clave al momento de llevar a cabo el proceso de desarrollo de software y teniendo en cuenta las características del proyecto presentado en este trabajo, se estableció como punto fundamental para contribuir a su desarrollo.
Lean Software Development considera la relación con el cliente como uno de sus cuatro valores principales, e incluso va un poco más allá: “colaboración con el cliente por encima de la negociación contractual”.Cuando se habla de metodologías ágiles solemos referirnos a metodologías que admiten el manifiesto ágil y a través de ellas se llega a descubrir formas

mejores de desarrollar software. A través del presente trabajo se ha aprendido a valorar a los individuos y sus interacciones por sobre los procesos y herramientas a aplicar. De esta manera, en general, en esta sección el objetivo es presentar como se asumieron los roles de los integrantes del equipo del proyecto teniendo en cuenta las características de la empresa.
La propuesta que se expone a continuación puede ser asumida en cualquier equipo de desarrollo y aunque cada producto y cada proyecto tienen su propia naturaleza, en el caso del presente trabajo, el papel del cliente se corresponde con uno de los integrantes del equipo de desarrollo, lo cual es una ventaja enorme ya que cada especificación y cada necesidad estaban en constante foco y eran perfectamente comprendidas para ser resueltas durante el proceso de desarrollo. De este modo se obtuvo feedback inmediato, con lo que se hizo más fácil la entrega deavances del desarrollo realizado.
En un aspecto general, en el proyecto se destacaron los siguientes aspectos:

- **Relación con el cliente**: como se mencionó previamente, la relación con el cliente tuvo características peculiares a raíz del doble rol de uno de los integrantes del equipo de desarrollo. Por esto, se tuvo acceso libre a cualquier tipo de información y recuso necesario para llevar a cabo la aplicación. Sin embargo, por momentos esta situación tuvo efectos contraproducentes en el equipo. El hecho de tener completamente incorporados los requerimientos y conocimientos del entorno de implementación, hizo poco fluida la comunicación en ocasiones. Esto se sobrellevó manteniendo reuniones de intercambio de ideas y conocimientos entre los integrantes del equipo en donde cada individuo expuso sus conocimientos más afianzados y respondió todas las preguntas del otro. De esta manera se logró tener un nivel de conocimiento equilibrado en los integrantes, tanto del proceso hilandero y teoría textil como de la metodología objeto de estudio y las herramientas a aplicar durante el desarrollo. Las reuniones fueron de carácter informal, pero siempre atentos a la posibilidad de surgimiento de nuevas ideas para aplicar al trabajo.
- **Simplicidad de la aplicación**: No fue necesario hacer flexibles partes del sistema ni adaptables a gustos y diferentes exigencias de posibles clientes ya que el entorno y usuaria final estaba estrictamente definido. Las herramientas que se buscaron para implementar fueron las más sencillas posibles, lo cual determinó un sistema simple y fácil de comprender y mantener, acorde a la premisa de eliminar cualquier gasto de trabajo o tiempo innecesario desde el punto de vista de la funcionalidad y que no aporte valor real al cliente. Otro punto que se tuvo en cuenta a la hora de elegir las herramientas y tecnología a utilizar fue la poca variabilidad del entorno, ya que la hilandera ha evolucionado mucho en todo lo referido a tecnologías de control de calidad y fabricación del hilado, pero el manejo de la materia prima se realiza de la misma manera desde hace décadas sin muchas perspectivas de cambio. Por lo tanto, un sistema que cumpla con las necesidades del manejo de fibra no cambiará de requisitos fácilmente y permanecerá estable y sin necesidad de modificaciones.

6.5.5. Planificación de las iteraciones

La planificación del proceso de desarrollo divido en iteraciones estuvo basado en el mapa de flujo de valor y graficado en el tablero kanban. Se establecieron las siguientes normas en función de las características del proyecto:

- Se determinó un tiempo de duración de cada iteración de entre dos semanas y cuatro semanas. Luego de trascurrido este plazo, los integrantes del equipo se deberían reunir para mostrar los avances de sus respectivas tareas y verificar el proceso en función del VSM y tablero kanban. Las iteraciones más largas expresadas en el mapa de valor se dividieron en sub iteraciones que se ajustaran a los tiempos establecidos previamente.
- En lo posible, luego de cada iteración se debería obtener una parte funcional del sistema en desarrollo.
- Ninguno de los integrantes podría realizar algún cambio sobre el trabajo del otro. Cada uno podría hacer sugerencias sobre la implementación, pero siempre en compañía del otro y en conciliación sobre la modificación.

Las iteraciones se planificaron haciendo uso de una herramienta visual llamada *OpenProj* que permitió el seguimiento visual de los tiempos que requeriría el proyecto. Esta herramienta es de código abierto y representó un solución al problema de costos y desperdicios que plantea la metodología de desarrollo Lean frente a otras herramientas de administración visual de licencia paga como por ejemplo *Microsoft Project*.

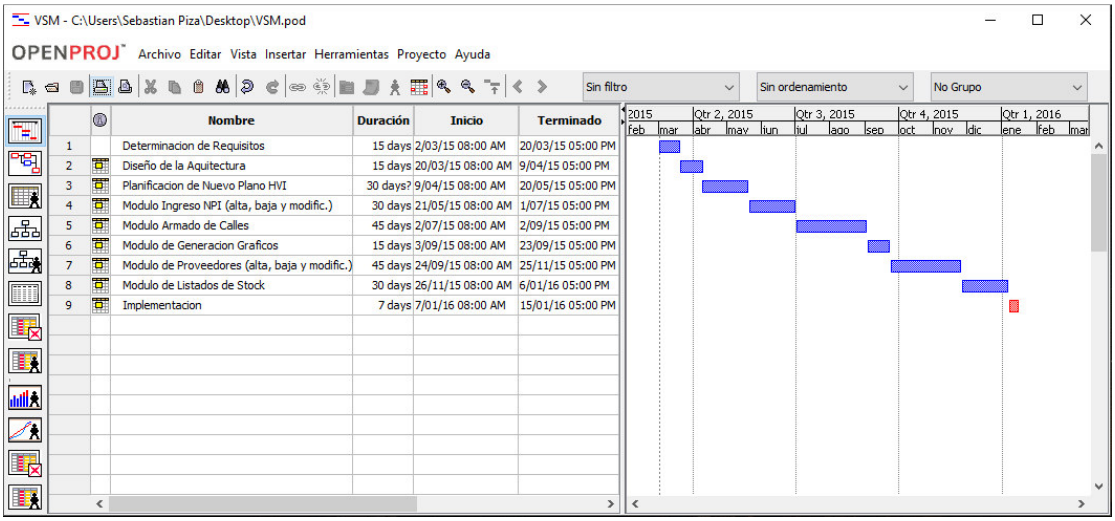


Figura 6.7 Planificación de Iteraciones con OpenProj

6.5.6. Uso de Formularios A3

En su origen, el formulario A3 fue desarrollado por Toyota como formato para desarrollar propuestas de mejora y su seguimiento, pero con el tiempo pasó a transformarse en una herramienta de resolución de problemas utilizada en las metodologías agiles y su aplicación puede darse en cualquier contexto. La idea de los formularios A3 es muy simple: comunicar un problema y su propuesta de mejora en una sola hoja. Esto permite a los miembros del equipo entender fácilmente cual es el problema y la solución propuesta.
En el formulario deben marcarse con claridad los siguientes puntos:

- Objetivo: identificar el problema o necesidad
- Situación actual: entender las circunstancias que generan la necesidad de resolver un problema.
- Situación deseada: se la plantea al analizar la causa raíz del problema.
- Acciones: medidas a tomar para desarrolla la mejora objetiva.
- Validación: contemplar la implementación de la mejora y reconocer resultados.

El formato del formulario varía en función a las preferencias de cada autor o usuario, pero siempre se respetan los puntos especificados anteriormente. Diferentes autores comentan que un problema constante que ocurre es la suposición de que hay un solo formato de formulario A3 que se debe utilizar, pero aunque siempre se encuentra valor en la normalización, la aplicación de una planilla estándar a todas las situación que pueden presentarse tiende a causar tantos problemas como los que pretende resolver.
El modelo usado para el desarrollo del presente trabajo fue extraído del sitio https://docs.google.com/presentation/d/1kksOO1dlCEdYeA8vcgdmhnL3rD8bfuF4JMW5qfDJmi0/edit#slide=id.p y posee el siguiente formato:

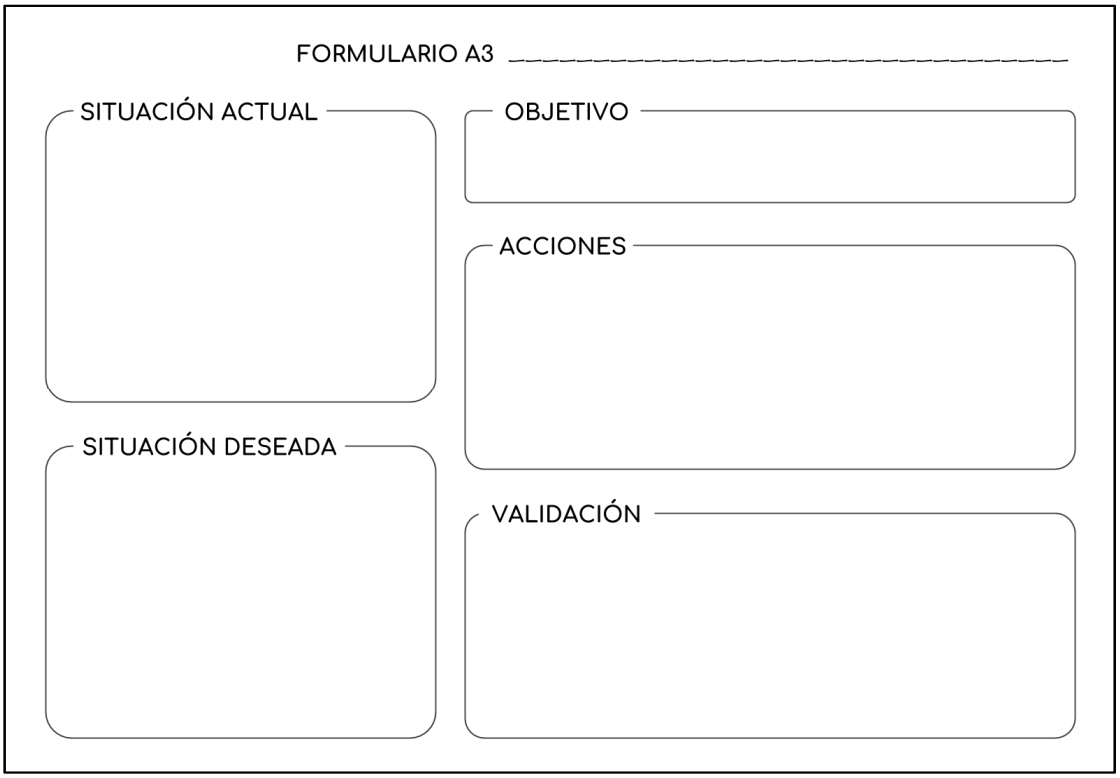


Figura 6.8 Formulario A3 genérico

En el libro "*Understanding A3 Thinking: A Critical Component of Toyota's PDCA Management System*" se definen siete elementos que deben estar presentes en la mentalidad de las personas que aplican la herramienta:

1. Proceso de pensamiento lógico orientado a la causa raíz.
2. Objetividad.
3. Conseguir resultados utilizando procesos excelentes.
4. Síntesis, destilación y visualización.
5. Alineación con todos los interesados.
6. Coherencia y consistencia a través de la organización.
7. Poseer una visión global de la contramedida.

Cada uno de estos puntos fueron tomados como referencia y se intentó adaptar la mentalidad para hacer un correcto uso de la herramienta.
A continuación se detallan los casos en los que fue utilizado el formulario A3 durante el proceso de desarrollo:

6.5.7. Primer Caso

En el presente formulario A3 se trabajó sobre el tema raíz que justifica el trabajo y que es la necesidad expresa de generar un nuevo sistema de armado de calles en Coteca. Se plantea

como objetivo la mejora en el armado de mezclas de algodón en función del análisis de la situación actual, en donde se especifican los puntos débiles del sistema actual:

- Su calidad de artesanal: el trabajo realizado, materializado en las mezclas de algodón, depende únicamente de la habilidad de la persona que la realiza.
- La falta de un criterio formal de selección de fardos: íntimamente relacionado al punto anterior. La selección de fardos se realiza únicamente en búsqueda de completar promedios de calidad, pero sin tener en cuenta variación de proveedores y de proporciones de calle a calle.
- La lentitud de todo el proceso: El armado de una sola calle podía tomar hasta más de una hora dependiendo de las condiciones de stock en las que se trabajaba.

De acuerdo a lo planteado, se proyecta un situación deseada en la que los puntos anteriormente marcados se solucionan y se obtiene un sistema de armado de calles formalizado, fácil de utilizar y rápido.

En la sección de acciones del formulario se plantean dos acciones resumidas y puntuales para abordar el objetivo: idear y plantear una nueva metodología de armado de calles y probar la misma antes de realizar cualquier tipo de codificación.

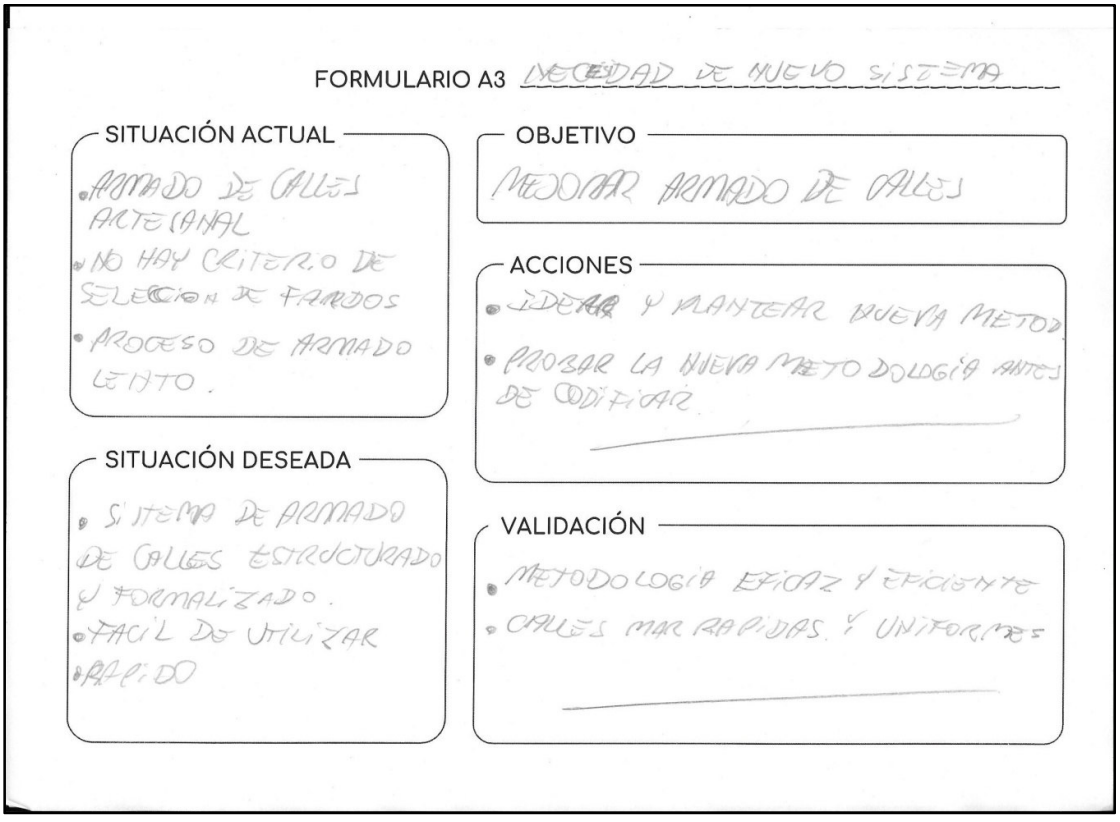


Figura6.9 Primer Caso de Formulario A3

Luego de un proceso de planificación y confección de un nuevo criterio de armado de calles, se probó el mismo en la planta simulando mezclas de algodón reales y comparando resultados con los obtenidos mediante el método anterior. La nueva técnica resultó ser

mucho más eficiente y se consideró apropiada para trabajar, lo que dio luz verde al comienzo de la planificación para el desarrollo del sistema.

6.5.8. Segundo Caso

Como siguiente paso a la mejora del sistema de armado de calles, se planteó la necesidad de mejorar el armado de los planos de los camiones con fibra que llegaban a la planta desde los proveedores.
En el formulario A3 se especifica la situación actual de manera clara y resumida. La situación deseada se plantea en términos más específicos respecto al desarrollo de la aplicación y su uso por parte del usuario.

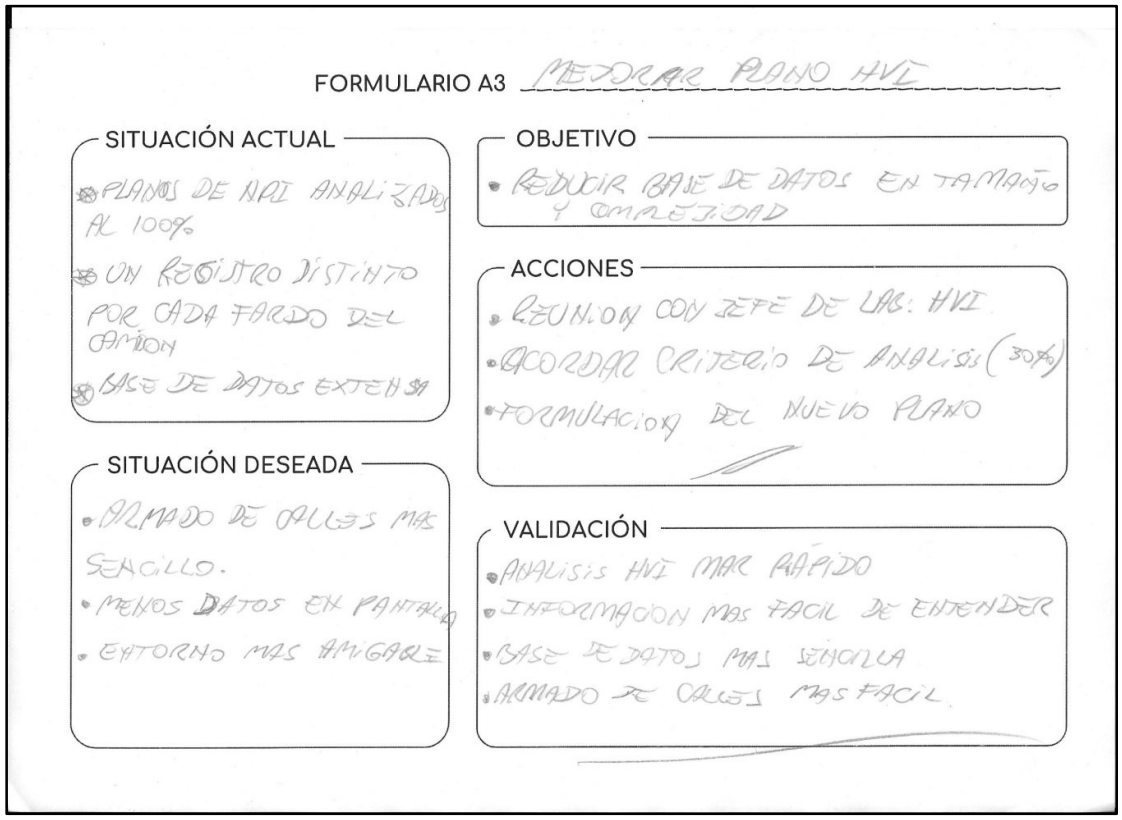


Figura 6.10 Segundo Caso de Formulario A3

Luego se planearon las acciones a llevar a cabo y al final se procedió a la validación de los resultados. El resultado de este segundo caso fue la restructuración del plano HVI a un formato más simple y marcó otro hito en la planificación del sistema para su desarrollo.

6.5.9. Tercer Caso

Para uno de los integrantes del equipo de trabajo el proceso textil y toda la maquinaria interviniente eran completamente desconocidos. Al momento de emprender un proceso de desarrollo de un sistema específico para este entorno, no tener conocimiento de la industria textil ni de los conceptos que se verían reflejados en la aplicación resulta un cuello de botella.
Una vez planteado el objetivo, la situación actual y la deseada, se llevaron a cabo visitas autorizadas y gestionadas en la empresa para la captura del conocimiento necesario para abordar los temas pertinentes al desarrollo de la aplicación. La empresa aportó incluso bibliografía a modo de préstamo para su estudio y uso de acuerdo a las necesidades que se presentaran.

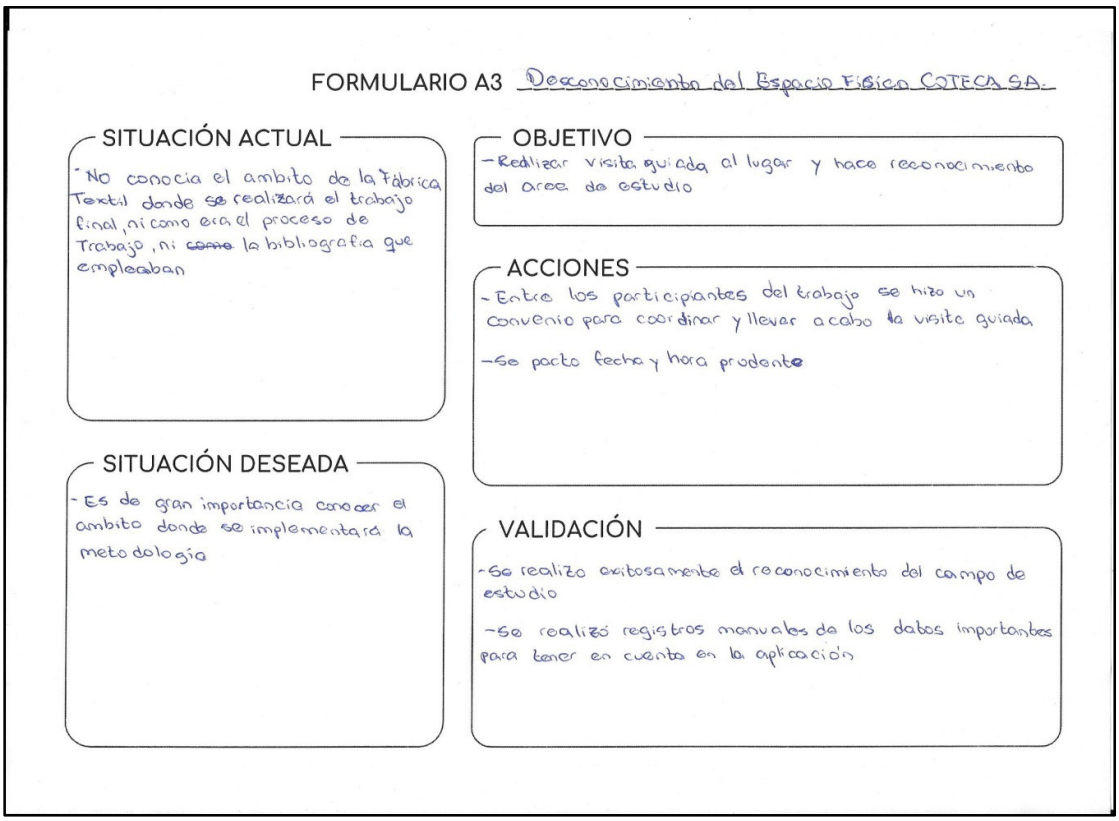


Figura 6.11 Tercer Caso de Formulario A3

6.5.10. Administración del código fuente

Se tomó el concepto de control de versiones desarrollado en el marco teórico de este trabajo y se ahondó sobre la factibilidad de aplicación en función de las características del mismo. La idea básicamente significa tener el código fuente, la información y datos importantes en un repositorio central que mantiene una versión de los mismos.

Para ellos se procedió a probar la herramienta *TortoiseSVN*, la cual es un sistema de control de versiones creado para que varios desarrolladores puedan trabajar en un proyecto de manera simultánea y de forma más o menos ordenada. Se debe tener servidor que sirva de repositorio en donde se centralicen los datos y allí se chequean los cambios que se van generando. El desarrollador hace las modificaciones y los sube al repositorio para que estén disponibles para los otros desarrolladores. Además se genera un registro de cambios donde se pueden colocar comentarios para brindar mayor información relacionada al trabajo realizado.

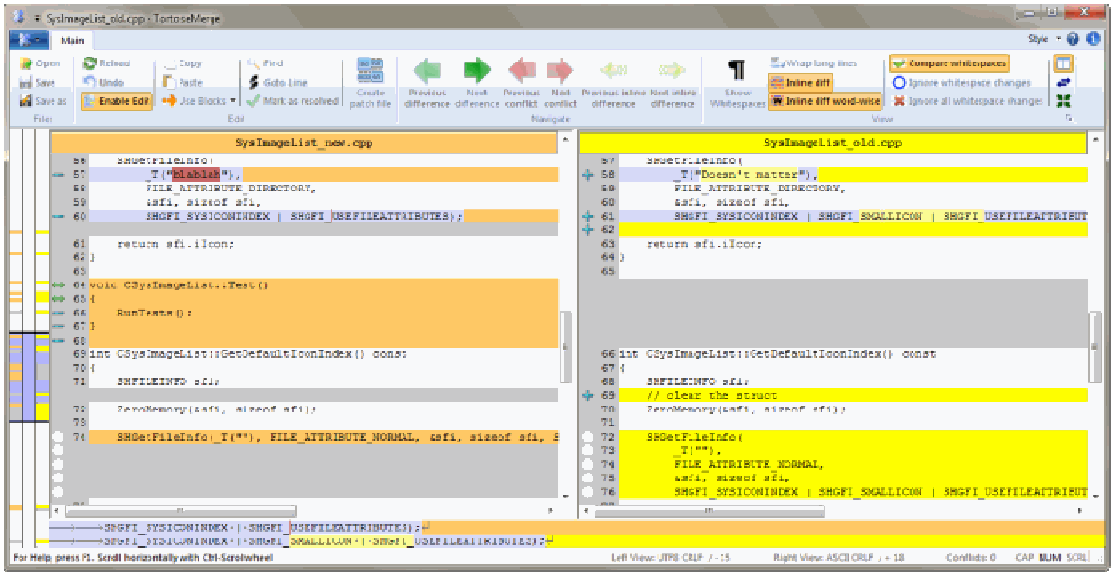


Figura 6.12 Entorno de TortoiseSVN

La creación del repositorio admite tres posibilidades: hacerlo de manera local (para proyectos en los que solamente hay un solo desarrollador en una sola computadora), utilizar un repositorio al que se acceda por red y por último montar un repositorio en un servidor web al que se puede tener acceso desde cualquier ubicación.
Se hizo evidente la necesidad de buscar un servidor para poder probar la herramienta de manera correcta y por lo tanto se buscó otra alternativa mucho más sencilla y sin requerimientos adicionales de funcionamiento.
Se utilizó una herramienta ampliamente difundida de alojamiento de archivos en la nube: *Dropbox*.
Pese a no ser un programa de control de versiones propiamente dicho, para el caso de este trabajo cumplió perfectamente con esa función de acuerdo a las características propias del desarrollo. Solamente dos personas podrían acceder a los archivos guardados en la nube y conservarían una copia de manera local. El control de cambios proporcionado automáticamente por *TortoiseSVN* se haría de manera sencilla a través de una comunicación informal en caso de ser necesario, ya que normalmente ambos integrantes del equipo trabajarían conjuntamente. La naturaleza sencilla del proyecto junto con la metodología estudiada proporcionaron los fundamentos para la búsqueda de herramientas sencillas y fáciles de aplicar, siempre con las premisas de eliminar cualquier actividad o

proceso que no genere valor y de optimizar el flujo de trabajo desde el punto de vista específico del desarrollo y de las personas.

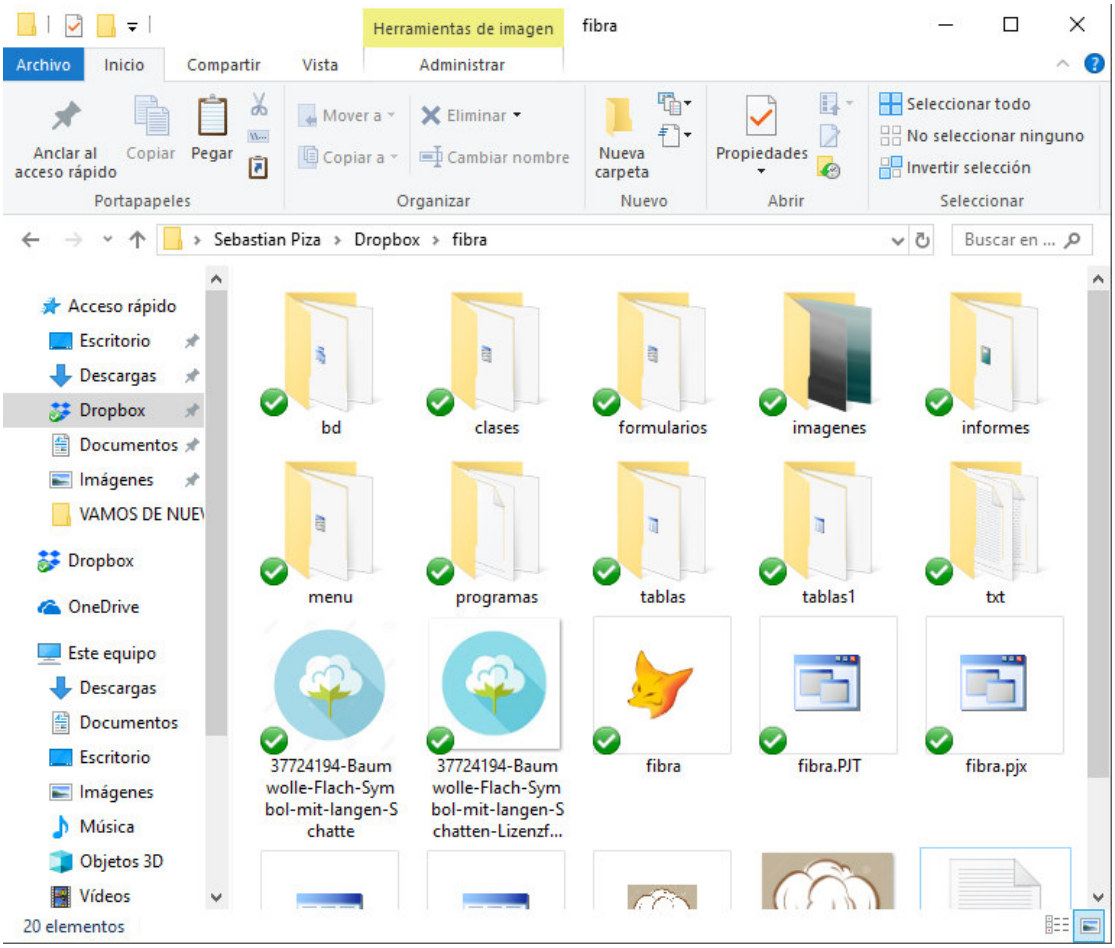


Figura 6.13 Repositorio en Dropbox

6.5.11. Desarrollo de la aplicación

Como herramienta de desarrollo se eligió Visual FoxPro 9 ya que se considera que su relativa simpleza permitirá desarrollar la aplicación más rápidamente, sobre todo por ofrecer ventajas como poseer su propia base de datos con un motor muy rápido, ser accesible para pequeñas empresas, fácil soporte y su aplicación en el entorno de Windows, lo que lo hace más familiar y predecible. Se hará una evaluación de la implementación de los principios y pensamiento del desarrollo de software Lean y al examinar el caso de estudio se logrará la comprensión de los diferentes pasos y procedimientos que intervienen en las actividades de implementación de Lean.

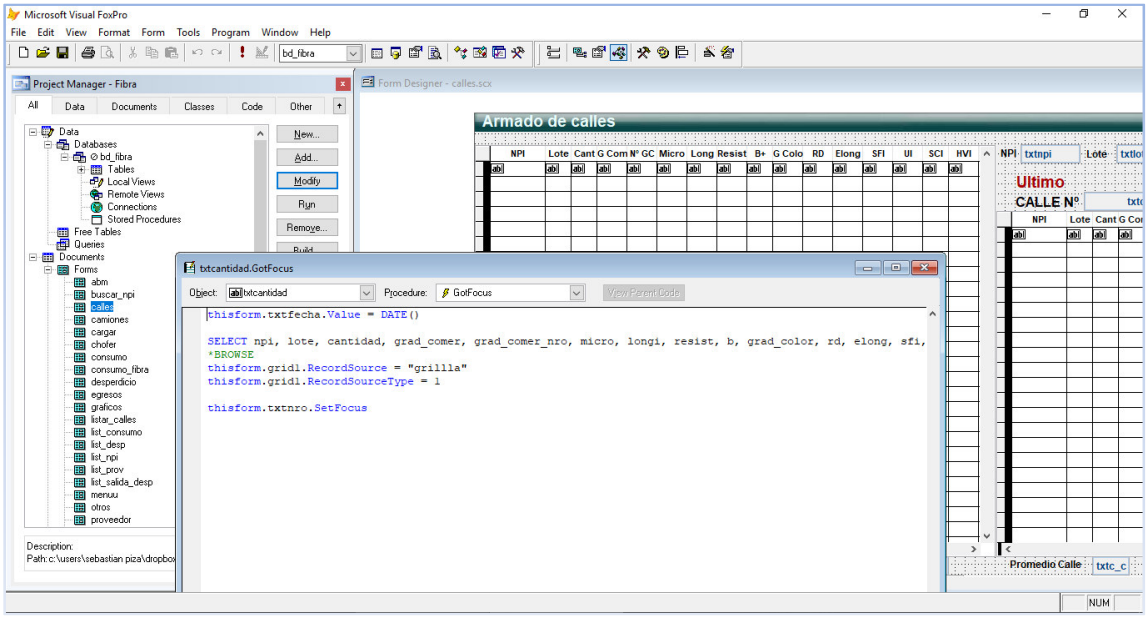


Figura 6.14 Diseño del Sistema en Fox

6.5.12. Análisis de datos

Una vez finalizado el desarrollo y recogidos los datos, se procede a estudiarlos mediante un análisis de contenido. Se trata de un examen en profundidad utilizando técnicas cuantitativas o cualitativas y que no se limita a los tipos de variables que puedan medirse o el contexto en el que se crean o presentan. En el análisis de contenido, el estudio de los datos se lleva a cabo reduciéndolos a categorías temáticas. En este trabajo los datos se dividen en tres categorías: costo, calidad y respeto. Esta división permite explorar la relación entre los conceptos y temas identificados. A lo largo del análisis se realiza la búsqueda de palabras específicas como tiempo, costo, calidad y respeto. Primero se centró la atención en los datos recogidos a través de la revisión de la bibliografía y elementos del marco teórico. Luego se analizan los datos empíricos de las entrevistas y del propio proceso de desarrollo utilizado como caso de estudio y al ser comparados con la información obtenida del marco teórico, se permite la discusión en torno a la validez de la teoría Lean, llegando así a poder lograr las conclusiones de este trabajo.

7. RESULTADOS ALCANZADOS

Esta sección, que será dividida en varias partes, incluye un análisis de los datos obtenidos de las entrevistas y del proceso de desarrollo de la aplicación. Esto se llevó a cabo en función del objetivo de validar la teoría Lean desde la óptica de la calidad, las personas y los costos. Se buscó identificar las posibles formas de medir y mejorar la productividad de los individuos participantes en proyectos de desarrollo de software y observar el papel que ha tenido la metodología Lean en la productividad de los procesos de trabajo. Las formas de medir y mejorar la productividad y la importancia de los paradigmas de Lean se basan en las opiniones y experiencia de los participantes de este estudio.

7.1. MEDICION Y MEJORA DE LA PRODUCTIVIDAD

Desde un punto de vista general, podríamos decir que medir la velocidad entre un esfuerzo de trabajo y otro es una manera eficaz de evaluar la productividad del equipo de desarrollo. La velocidad se puede medir en puntos de historia relativos, puntos de funciones por hora, etc. Se estuvo de acuerdo en que esto debería calcularse periódicamente y ser reflejado en gráficas para ver la tendencia resultante. La medición de la productividad basada en el número de puntos de función se alinea con los estudios de Symons (1988), Maxwell y Forselius (2000) y Ferrão (2010), que mantienen conteos de puntos de función como una manera óptima de medir la productividad. Sin embargo, estos investigadores atribuyen importancia a la cantidad de líneas de código por programador como una medida óptima de la productividad. Se puede ver fácilmente que el conteo de puntos de historia en la actualidad es más eficaz que contar líneas de código por programador. Durante el desarrollo del trabajo, no se consideró como medidas ideales de productividad a la documentación, cambios implementados o cualquier otra medida utilizada por filosofías más duras y tradicionales.

En lugar de ello, podemos afirmar que medir el rendimiento de un equipo en función de su funcionamiento en un momento anterior en el tiempo es otra manera de observar en qué medida se mejora la productividad de los procesos de trabajo. De esto se desprende la conclusión de que es importante no comparar el rendimiento de dos equipos, siempre y cuando no estén trabajando en las mismas condiciones.

El número de características entregadas durante un periodo determinado y la cantidad de tiempo que tarda una característica en pasar por varios cambios son otras medidas de la productividad del equipo. Se dedujo que la duración del tiempo de transición que necesita una función para pasar de “en progreso” a “hecha” es un mejor indicador que contar bugs, horas y comparar estimaciones con datos reales. Esto parece contraponerse a la importancia que normalmente se da a la comparación entre lo planificado y las actividades reales (en términos de tiempo, costo, horas-persona/semana, etc.), la varianza y el análisis del valor acumulado. En cierta manera, este punto de vista contradice también la idea de considerar como una medida importante de productividad al tiempo necesario para corregir un error, ya que muestra la madurez del flujo de trabajo.

Muchas organizaciones miden el rendimiento del equipo mediante el control de la medida en que se cumplen los hitos fijados en un principio. Se tiene un proceso para establecer varios

hitos para un paquete de trabajo (o ciclo de vida de desarrollo del producto): preparación, inicio formal, compromiso adquirido, iteraciones realizadas y cerradas. Para cada etapa hay ciertos checklists para asegurarse de que todos los criterios se cumplen.
La observación y medición de la calidad también es una manera de medir el rendimiento del equipo en busca de la mejora continua. La calidad se basa en tres parámetros: el primero es el nivel de "retrabajo" (la relación entre reparar y construir, que es la cantidad de tiempo que se gasta en la reparación de una tarea que llevó una cierta cantidad de tiempo en llevarse a cabo). Es importante señalar que el número de defectos no es una medida objetiva, ya que se puede comparar dos equipos que enfrentaron el mismo número de errores, pero es difícil comparar el nivel de los respectivos tamaños de los problemas. El segundo parámetro de calidad es el cambio. Si los requisitos iniciales son claros, no debería haber ningún cambio que frene el avance de los trabajos. Mientras que la relación de reparación revela la calidad del trabajo realizado por el equipo de desarrollo, la relación de cambio revela la calidad de los requisitos producidos. Estas medidas se pueden reflejar en una tarjeta de puntuación y colocarla en un panel de control o pizarra, de manera que esté disponible para todo el mundo y ayude a medir periódicamente el rendimiento del equipo.
Además de las medidas antes mencionadas, existen valores más sutiles. Como ejemplo, valdría la pena mencionar la intuición de los jefes de equipo; la manera en que se sienten respecto al trabajo realizado por los miembros del equipo. Por otro lado, se podría afirmar que el estado de ánimo del equipo revela los problemas antes que la estadística y las métricas. Si la moral del equipo cae y no se hace nada para mejorarla, esto se verá reflejado en tiempos de ciclo más largos y ritmos de trabajo más lentos. Por lo tanto, es muy importante colaborar estrechamente con los compañeros de equipo y observar los cambios.
Otro punto muy importante es el hecho de conservar la satisfacción del cliente como un factor crucial a considerar cuando se apunta a mejorar la productividad; esto se debe a que los clientes felices tienden a lograr un incremento en el valor del negocio. Esto puede lograrse mediante el establecimiento de un contacto constante con el cliente durante el avance con el fin de desarrollar y modificar el código, alineándolo con sus necesidades y dejando que intervenga y provea retroalimentación antes del lanzamiento final.
La fase de reflexión sobre las lecciones aprendidas al final de cada lanzamiento se considera crucial para la mejora de la productividad de acuerdo a la experiencia durante el proceso de desarrollo. Se puede ver fácilmente que las medidas de productividad que los participantes de este estudio proporcionan son de cierta manera diferentes de lo que sugiere la literatura. Varias investigaciones han definido a la productividad como la funcionalidad, la complejidad y la calidad de los productos de software o como el tiempo y el costo requeridos para entregar los productos. En el estudio concreto, como se ha observado anteriormente, la calidad es un factor clave a considerar, pero además existen otros que también son medidas de productividad y han sido utilizados por los participantes del trabajo.
Tomando en cuenta la relativa experiencia de trabajo personal, los participantes de este estudio mantienen que es difícil medir la productividad cuando se pretende evaluar la eficiencia con el fin de aumentar el valor del negocio. Dada la naturaleza intangible del software, se hace muy difícil de medir.

7.2. CICLO DE VIDA DEL DESARROLLO DE SOFTWARE

En lo referido a la longitud del ciclo de desarrollo, surgieron diferentes opiniones con respecto a su impacto en la productividad. Hasta cierto punto se estuvo de acuerdo en el hecho de que los ciclos de desarrollo más cortos son los mejores, ya que se traducen en menos tiempo para llegar al cliente, lo que en otras palabras significa una entrega más rápida, retroalimentación más temprana y como resultado una mejora en el valor del negocio.Los ciclos cortos ayudan a comprobar continuamente si se están cumpliendo los hitos de un proyecto y cambiar de dirección si los resultados producidos no son los esperados. Otra ventaja que se puede señalar es que un ciclo de desarrollo más corto se traduce también en la recuperación más rápida de la inversión.
Se determinó que si se construye la funcionalidad más valiosa en primer lugar y se libera tan pronto como sea posible, esta funcionalidad puede comenzar a brindar frutos lo más pronto posible también. A menudo el valor de esta primera versión es suficiente para compensar el costo del resto del proyecto, por lo tanto se puede llegar a recuperar la inversión en un mes en vez de en un año. Además, al acortar el tiempo del ciclo sin reducir el tamaño del incremento, se está haciendo la misma cantidad de trabajo efectivo en un tiempo más corto, lo que por definición aumentaría la productividad.
A pesar de los beneficios de los ciclos de desarrollo más cortos, es claro que su aplicación representa un desafío. Un equipo que decide producir código diariamente puede ser muy productivo, pero debe ser maduro y tener una excelente capacidad de planificación para organizar las tareas.
Por otro lado, los ciclos de desarrollo largos requieren más esfuerzos de planificación y están asociados al riesgo de avanzar demasiado lejos en la dirección equivocada si no se realizan controles durante un largo tiempo.
Durante la realización de este trabajo se pudo notar que lo que se consideraba óptimo era un ciclo de desarrollo con una longitud de entre dos y cuatro semanas. Sin embargo, la duración de un ciclo depende de la complejidad de la función y de la existencia de otros equipos con los cuales poder interactuar (en el caso de grandes proyectos con equipos dispersos), coordinando y alineando las actividades con el fin de garantizar una colaboración eficaz.
Independientemente de todo lo planteado respecto al tema, existen opiniones que afirman que los ciclos de desarrollo más rápidos no necesariamente resultan en una mejor calidad y mayor productividad. Podemos pensar en proyectos en los que la calidad de las etapas no es lo suficientemente alta o proyectos en los que los marcos están mal hechos. En algunos casos parece que el análisis y pensamiento se retrasan hasta después del desarrollo. Algunas veces puede parecer que un proyecto entrega rápidamente software funcional, pero luego de algunas iteraciones el sistema se vuelve más complejo y se descubre que se tomaron decisiones equivocadas y solucionar esto retrasa el proyecto. Un efecto secundario de esto es que los equipos se mantienen corrigiendo y refactorizando el código en lugar de desarrollar nuevas funcionalidades.
Ninguno de los textos consultados para la realización de este trabajo fue enfocado en los efectos negativos de ejecutar ciclos de desarrollo muy cortos. Por el contrario, muchos argumentan que los clientes no esperarán al demorarse el producto, ya que los retrasos pueden ser destructivos tomando en cuenta que el mercado del software ofrece siempre alternativas de cualquier tipo de aplicación.

Investigadores como Livermore (2008) y Cordeiro (2008) mantienen que la longitud óptima de una iteración es de una a cuatro semanas, la cual es una afirmación que resultó ser cierta en la práctica a la hora de juzgar la respuesta brindada por los involucrados en este trabajo.

7.3. TAMAÑO DEL EQUIPO

Al discutir sobre el tamaño ideal del equipo se hizo mucho hincapié en la bibliografía, ya que no se contaba con experiencia respecto al desarrollo de grandes proyectos que requirieran gran cantidad de participantes. Teóricamente, sería bueno que los equipos estuvieran conformados por siete +/- dos personas. El tamaño ideal varía ligeramente de una fuente a otra. Algunas están de acuerdo en tener de cuatro a ocho personas por equipo, otras no más de siete, otras de seis a nueve, mientras que otras ven que más de ocho es un tamaño efectivo. A pesar de las pequeñas variaciones, todas coinciden en afirmar las ventajas de los equipos de tamaño medio, lo que indica que no debe ser demasiado pequeño ni tampoco grande. Esto se alinea sobre todo con un estudio realizado por Schwaber y Sutherland (2011), quienes ven a los equipos de tamaño medio como los más eficientes.
En el caso de equipos pequeños (como el que realizó el presente trabajo), resulta complejo mantenerse dentro de una estructura formal de desarrollo y ajustarse estrictamente a una metodología. Se pudo deducir que los equipos compuestos por solo unos pocos individuos no tienen una discusión efectiva, ya que no es posible proporcionar muchos ángulos diferentes para una lluvia de ideas y juzgar determinadas situaciones durante el trabajo diario. Schwaber y Sutherland (2011) afirman lo mismo en su estudio cuando afirman que los equipos muy pequeños disminuyen la velocidad y la productividad del desarrollo.
Sin embargo, podríamos señalar que los equipos pequeños pueden brindar ciertas ventajas, como ser más fácilmente auto-regulables y organizarse a sí mismos, mejorando en teoría la velocidad de desarrollo de manera notable.
En resumen, en el caso de los equipos grandes, se puede encontrar dificultad en conseguir autonomía y auto-organización debido a su gran número de miembros. En estas circunstancias, se señala que el tamaño del equipo y el número de canales de comunicación están conectados directamente entre sí. Una gran cantidad de miembros causa demasiada sobrecarga dentro del equipo y lleva a la creación de pequeños grupos dentro del equipo original, disminuyendo su cohesión y conduciendo a una menor productividad. Otra dificultad que pueden experimentar los equipos de gran tamaño es encontrar un espacio físico donde encajar. Resulta beneficioso que el equipo comparta un mismo ámbito de trabajo, ya que les resulta más fácil comunicarse y colaborar entre sí. Una vez más, los resultados se apoyan en el estudio de Schwaber y Sutherland (2011), que afirma que los equipos muy grandes (con más de nueve miembros) requieren esfuerzos especiales para coordinar y gestionar su trabajo.
Otro punto de vista que se puede adoptar es omitir la búsqueda del tamaño ideal del equipo y ajustar la cantidad de miembros dependiendo del volumen de trabajo que se deba realizar. En este caso se debería incluir la cantidad necesaria de personas competentes que posean entre ellos las habilidades suficientes para formar un grupo viable que sea capaz de hacer frente a las complejas exigencias del trabajo. Un equipo compuesto por individuos calificados puede mejorar la velocidad del desarrollo y la productividad, como así también reducir el riesgo programado asociado al proceso de desarrollo.

7.4. EL PAPEL DE LOS ENFOQUES AGILES EN LA MEJORA DE LOS PROCESOS DE TRABAJO

Si se indaga sobre el aporte de las metodologías agiles a la mejora de los procesos de trabajo, probablemente se llegará a la conclusión de la superioridad de estos enfoques sobre los tradicionales. El uso de metodologías agiles se traduce en entrega más rápida, continua interacción con el cliente y una mayor calidad del trabajo, cambiando la mentalidad de la empresa y ayudando a resolver los obstáculos y afianzando el éxito. Incluso investigadores afirman que todos estos aspectos positivos mejoran la moral organizacional de las empresas que las aplican. Sin embargo, las metodologías agiles no necesariamente aumentan la velocidad de desarrollo, pero mejoran la calidad y ayudan a entregar valor al cliente, permitiendo afinar los procesos, métodos y la forma de trabajo. La incorporación de metodologías agiles hace que las empresas se conviertan en "organizaciones de aprendizaje" e integren al equipo de desarrollo dentro del proceso como un todo, haciéndolos capaces de enfrentar los cambios, hacer retrospectiva y buscar la mejora continua.
Toda la bibliografía referida a metodologías agiles que se ha utilizado para construir el marco teórico de este trabajo, hace hincapié en su superioridad frente a los enfoques tradicionales. Sin embargo, podríamos discrepar con esta comparación, ya que consideraríamos que dependiendo de la situación, ambas son útiles y beneficiosas para el equipo de desarrollo. Son dos enfoques diferentes y a la vez fuertes en sus propios entornos. El uso de metodologías agiles haría una diferencia solo en aquellos equipos que continuamente están enfrentándose al cambio. Si los requisitos del cliente son perfectamente claros desde el principio, los métodos tradicionales (como el de cascada) serían eficaces en la misma medida que los agiles. De hecho, las metodologías tradicionales podrían incluso considerarse apropiadas para aquellos proyectos que se caracterizan por tener requerimientos de negocio estáticos, aun cuando la literatura señala que las mismas se encuentran sobrecargadas con documentación y procedimientos.
Independientemente de lo expresado en el párrafo anterior, es fundamental señalar las ventajas de implantar métodos agiles, enfocándonos en la flexibilidad de la estructura y la visibilidad de los actores que conducen a una gestión efectiva del riesgo y procesos de toma de decisiones, cuyos beneficios no pueden ser pasados por alto.

7.5. LOGRAR AFIANZAR EL RESPETO POR LAS PERSONAS

El concepto de respeto por las personas suena algo nebuloso, pero incluye acciones concretas y cambios culturales que reflejan ampliamente el respeto y la sensibilidad a la moral, no permitiendo que las personas desperdicien su trabajo, fomentando el trabajo en equipo y el desarrollo de personas con habilidades. Se pretende humanizar el trabajo y el entorno, logrando que sea limpio y seguro en un marco de integridad filosófica alrededor del equipo.
Durante el proceso de desarrollo de la aplicación que cumple el rol de caso de estudio, se estudiaron diversas fuentes a fin de poder aplicar o simular, en la medida de lo posible, distintos conceptos que apuntan a lograr afianzar el pilar fundamental de Lean que estamos tratando. A continuación se detallan los conceptos aplicados:

No molestar al cliente: se tomó como base la idea de que llamamos cliente a cualquier persona que consuma el trabajo o las decisiones del equipo de desarrollo. A partir de esto, se procuró analizar y cambiar constantemente los lineamientos del desarrollo a fin de evitar generar preocupaciones en los clientes, pero también cuidando que los miembros del equipo no se vieran forzados a desperdiciar su trabajo. En todo momento se mantuvo el objetivo de no entregar defectos, no generar esperas y no sobrecargar al equipo. Todos estos conceptos se relacionan estrechamente con el principio de eliminar el desperdicio.
Desarrollar a las personas para luego construir el producto: se buscó adquirir una mentalidad enfocada en la idea de que los integrantes del equipo pueden ser maestros los unos con los otros, aportado cada uno experiencias y conocimientos que puedan ser aprovechados por sus compañeros. De esta manera se logró consolidar la buena práctica de analizar la raíz de los problemas y hacerlos visibles, descubriendo las maneras de mejorar continuamente.
Los equipos y las personas deben desarrollar sus propias prácticas y mejoras: los integrantes del equipo afrontaron el desafío de cambiar y preguntarse constantemente que se podría mejorar; así se adquirieron habilidades de reflexión y resolución de problemas para decidir de manera más eficaz qué mejoras realizar.
Predicar con el ejemplo: el del grupo de trabajo comprende y actúa sobre el objetivo de "eliminar residuos" y "mejora continua" con sus propias acciones (y el resto del equipo ve eso).
Construir compañeros y desarrollar equipos: se debe apuntar a formar relaciones fuertes basadas en la confianza y brindar ayuda a los miembros del equipo para mejorar y seguir siendo rentables, generando una cultura de trabajo en equipo, no de trabajo en grupo.

7.6. LOGRAR LA MEJORA CONTINUA

La mejora continua está basada en varias ideas, las cuales se pusieron en práctica en la medida de lo que permitió el alcance del proyecto del que trata este trabajo. Los conceptos mencionados son:

7.6.1. GoSee

En un principio que no se encuentra en muchas culturas de gestión. Se trata de ir a la fuente, al lugar donde se encuentra el trabajo de real valor, para comprobar hechos y tomar decisiones correctas, crear consenso y alcanzar los objetivos a nuestra mejor velocidad. Puede ser considerado un principio crítico y fundamental y se destaca como el primer factor para el éxito de la mejora continua.
En la cultura del pensamiento Lean, todas las personas (especialmente gerentes y directivos de alto nivel) no deben pasar todo su tiempo en oficinas separadas o salas de reuniones, recibiendo información a través de reportes, computadoras y reuniones informativas.
En lugar de esto, para saber lo que está pasando realmente y ayudar a mejorar (eliminando la distorsión que proviene de la información indirecta), a menudo se debería ir al lugar del trabajo real para ver y entender por sí mismo lo que está pasando.
En el caso particular de este trabajo, el concepto de *GoSee* fue puesto en práctica concurriendo al lugar físico donde el Sistema de Gestión Técnica de Fibra sería aplicado. Se

familiarizó al equipo con toda la línea del proceso productivo de la empresa para lograr crear un efecto de sentido en las personas. Entender realmente cómo funciona el proceso para el cual se está desarrollando facilitó enormemente la interpretación de los conceptos técnicos que fueron necesarios aplicar al momento de la construcción, mejorando la toma de decisiones y afianzando la relación entre los integrantes del equipo de trabajo.

Figura 7.1 COTECA S.A. – Sector Peinadoras

Figura 7.2 COTECA S.A. – Sector Mecheras

Figura 7.3 COTECA S.A. – Playa de fardos (materia prima)

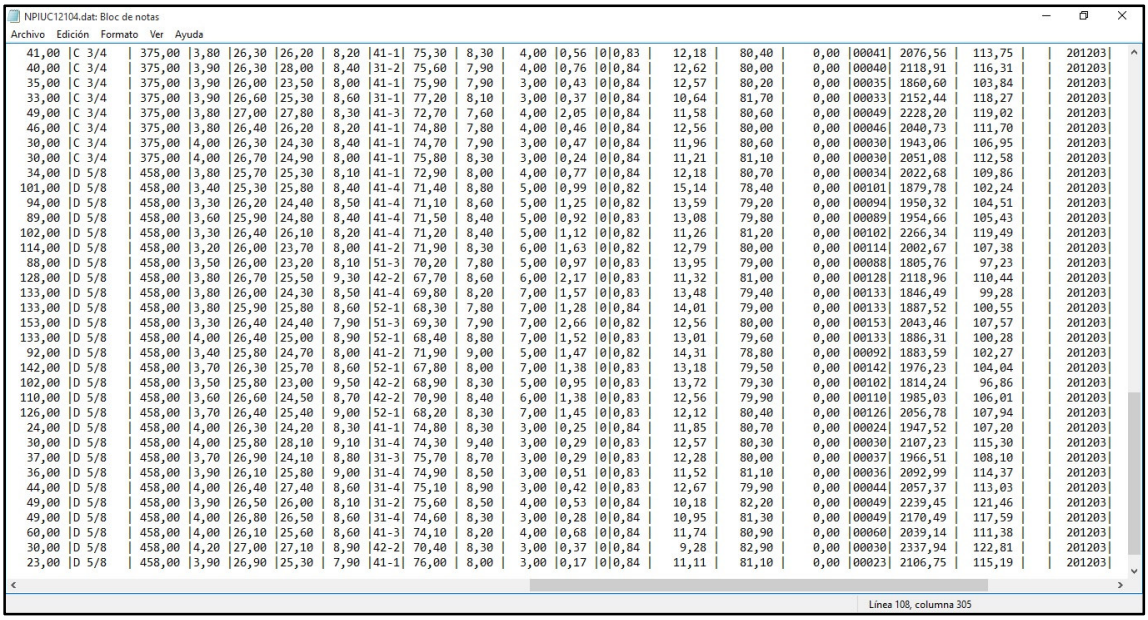
7.6.2. Kaizen

Este término se traduce a veces simplemente como “mejora continua”, pero eso hace que se confunda con el concepto más amplio que lleva el mismo nombre y por lo tanto no captura todo el significado del mismo.
Kaizen es una manera de pensar y al mismo tiempo una práctica. Como una mentalidad, sugiere que “*mi trabajo es hacer mi trabajo y mejorarlo*” y “*mejorar continuamente por nuestro propio bien*”. Más formalmente como práctica, Kaizen implica:

- Elegir y practicar técnicas que el equipo se ha comprometido a tratar hasta que hayan sido bien entendidas, es decir, estandarizar el trabajo.
- Experimentar hasta encontrar una mejor manera.
- Repetir para siempre.

Kaizen es aplicado a menudo mediante la repetición de eventos; sería deseable una cadencia regular y frecuente, como diariamente o semanalmente. En términos generales, un evento Kaizen está formado por dos etapas: analizar alguna situación actual hasta entenderla bien y diseñar experimentos para mejorarla. Durante este análisis y diseño, el foco debe centrarse en las actividades además de sentarse alrededor de una mesa y hablar. Una herramienta que resultó de mucha utilidad fue diagramar actividades en una pizarra y a partir de las ideas expresadas, discutir y buscar mejoras a las situaciones planteadas.

Un ejemplo de evento Kaizen durante el proceso de desarrollo fue la optimización del manejo de los archivos de texto denominados “planos de NPI”. Todos los camiones que ingresan a la planta están formados por fardos de algodón cuyas cantidades varían de 110 a 140 unidades, con un peso promedio de carga de aproximadamente 25.000 Kg. Cada uno de estos fardos es tomado como unidad mínima de trabajo en el proceso productivo y posee valores de calidad (físicos y químicos) únicos. A cada camión (llamado internamente NPI) se le asocia un archivo de texto generado por los instrumentos de análisis que van determinando los valores de calidad de cada fardo. Cada archivo contaba con una línea por cada fardo (120 fardos en el camión, 120 líneas en el plano), lo cual al ser trasladado al stock total que regularmente maneja la planta, podía generar una base de datos de hasta 15.000 registros. Hasta ese momento, no había nada llamativo en la situación, pero al comenzar a analizar la estructura de los planos fue notorio que los datos de cada línea eran muy similares dentro de un mismo NPI. Se inició la etapa de análisis de todo evento Kaizen y se confirmó y fundamentó la observación en base a los datos técnicos aportados por el personal de laboratorio de la empresa. La situación observada era de una inmensa redundancia en los datos que conformaban la base de fardos de la planta.Se buscó la manera de mejorar esto en base a reuniones del equipo de trabajo con el personal técnico de la empresa y se diseñó una nueva estructura del plano basada en lotes de fardos en función de coeficiente de variación de los parámetros clave. Se experimentó y se crearon archivos con registros que contenían el valor promedio de los distintos parámetros de calidad de los fardos cuyo coeficiente de variación de ciertos campos clave no fuera superior al 5%. Esto determinó que ese valor fuese muy representativo de la población, manteniendo controlada la dispersión en función del CV%. Como resultado, en vez de planos con más de 100 líneas, se obtuvo archivos con una cantidad que variaba entre una y cuatro líneas, y por ende una base de datos del stock de menos de 400 registros en el caso más extremo, con todas las ventajas que una base relativamente pequeña puede ofrecer y sin haber afectado de ninguna manera el trabajo real.


NPIUC12104.dat: Bloc de notas

Archivo Edición Formato Ver Ayuda

```
 41,00 |C 3/4  | 375,00 |3,80 |26,30 |26,20 | 8,20 |41-1| 75,30 | 8,30 |  4,00 |0,56 |0|0,83 |   12,18 |   80,40 |    0,00 |00041| 2076,56 |  113,75 |   | 201203|
 40,00 |C 3/4  | 375,00 |3,90 |26,30 |28,00 | 8,40 |31-2| 75,60 | 7,90 |  4,00 |0,76 |0|0,84 |   12,62 |   80,00 |    0,00 |00040| 2118,91 |  116,31 |   | 201203|
 35,00 |C 3/4  | 375,00 |3,90 |26,00 |23,50 | 8,00 |41-1| 75,90 | 7,90 |  3,00 |0,43 |0|0,84 |   12,57 |   80,20 |    0,00 |00035| 1860,60 |  103,84 |   | 201203|
 33,00 |C 3/4  | 375,00 |3,90 |26,60 |25,30 | 8,60 |31-1| 77,20 | 8,10 |  3,00 |0,37 |0|0,84 |   10,64 |   81,70 |    0,00 |00033| 2152,44 |  118,27 |   | 201203|
 49,00 |C 3/4  | 375,00 |3,80 |27,00 |27,80 | 8,30 |41-3| 72,70 | 7,60 |  4,00 |2,05 |0|0,84 |   11,58 |   80,60 |    0,00 |00049| 2228,20 |  119,02 |   | 201203|
 46,00 |C 3/4  | 375,00 |3,80 |26,40 |26,20 | 8,20 |41-1| 74,80 | 7,80 |  4,00 |0,46 |0|0,84 |   12,56 |   80,00 |    0,00 |00046| 2040,73 |  111,70 |   | 201203|
 30,00 |C 3/4  | 375,00 |4,00 |26,30 |24,30 | 8,40 |41-1| 74,70 | 7,90 |  3,00 |0,47 |0|0,84 |   11,96 |   80,60 |    0,00 |00030| 1943,06 |  106,95 |   | 201203|
 30,00 |C 3/4  | 375,00 |4,00 |26,70 |24,90 | 8,00 |41-1| 75,80 | 8,30 |  3,00 |0,24 |0|0,84 |   11,21 |   81,10 |    0,00 |00030| 2051,08 |  112,58 |   | 201203|
 34,00 |D 5/8  | 458,00 |3,80 |25,70 |25,30 | 8,10 |41-1| 72,90 | 8,00 |  4,00 |0,77 |0|0,84 |   12,18 |   80,70 |    0,00 |00034| 2022,68 |  109,86 |   | 201203|
101,00 |D 5/8  | 458,00 |3,40 |25,30 |25,80 | 8,40 |41-4| 71,40 | 8,80 |  5,00 |0,99 |0|0,82 |   15,14 |   78,40 |    0,00 |00101| 1879,78 |  102,24 |   | 201203|
 94,00 |D 5/8  | 458,00 |3,30 |26,20 |24,40 | 8,50 |41-4| 71,10 | 8,60 |  5,00 |1,25 |0|0,82 |   13,59 |   79,20 |    0,00 |00094| 1950,32 |  104,51 |   | 201203|
 89,00 |D 5/8  | 458,00 |3,60 |25,90 |24,80 | 8,40 |41-4| 71,50 | 8,40 |  5,00 |0,92 |0|0,83 |   13,08 |   79,80 |    0,00 |00089| 1954,66 |  105,43 |   | 201203|
102,00 |D 5/8  | 458,00 |3,30 |26,40 |26,10 | 8,20 |41-4| 71,20 | 8,40 |  5,00 |1,12 |0|0,82 |   11,26 |   81,20 |    0,00 |00102| 2266,34 |  119,49 |   | 201203|
114,00 |D 5/8  | 458,00 |3,20 |26,00 |23,70 | 8,00 |41-2| 71,90 | 8,30 |  6,00 |1,63 |0|0,82 |   12,79 |   80,00 |    0,00 |00114| 2002,67 |  107,38 |   | 201203|
 88,00 |D 5/8  | 458,00 |3,50 |26,00 |23,20 | 8,10 |51-3| 70,20 | 7,80 |  5,00 |0,97 |0|0,83 |   13,95 |   79,00 |    0,00 |00088| 1805,76 |   97,23 |   | 201203|
128,00 |D 5/8  | 458,00 |3,80 |26,70 |25,50 | 9,30 |42-2| 67,70 | 8,60 |  6,00 |2,17 |0|0,83 |   11,32 |   81,00 |    0,00 |00128| 2118,96 |  110,44 |   | 201203|
133,00 |D 5/8  | 458,00 |3,80 |26,00 |24,30 | 8,50 |41-4| 69,80 | 8,20 |  7,00 |1,57 |0|0,83 |   13,48 |   79,40 |    0,00 |00133| 1846,49 |   99,28 |   | 201203|
133,00 |D 5/8  | 458,00 |3,80 |25,90 |25,80 | 8,60 |52-1| 68,30 | 7,80 |  7,00 |1,28 |0|0,84 |   14,01 |   79,00 |    0,00 |00133| 1887,52 |  100,55 |   | 201203|
153,00 |D 5/8  | 458,00 |3,30 |26,40 |24,40 | 7,90 |51-3| 69,30 | 7,90 |  7,00 |2,66 |0|0,82 |   12,56 |   80,00 |    0,00 |00153| 2043,46 |  107,57 |   | 201203|
133,00 |D 5/8  | 458,00 |4,00 |26,40 |25,00 | 8,90 |52-1| 68,40 | 8,80 |  7,00 |1,52 |0|0,83 |   13,01 |   79,60 |    0,00 |00133| 1886,31 |  100,28 |   | 201203|
 92,00 |D 5/8  | 458,00 |3,40 |25,80 |24,70 | 8,00 |41-2| 71,90 | 9,00 |  5,00 |1,47 |0|0,82 |   14,31 |   78,80 |    0,00 |00092| 1883,59 |  102,27 |   | 201203|
142,00 |D 5/8  | 458,00 |3,70 |26,30 |25,70 | 8,60 |52-1| 67,80 | 8,00 |  7,00 |1,38 |0|0,83 |   13,18 |   79,50 |    0,00 |00142| 1976,23 |  104,04 |   | 201203|
102,00 |D 5/8  | 458,00 |3,50 |25,80 |23,00 | 9,50 |42-2| 68,90 | 8,30 |  5,00 |0,95 |0|0,83 |   13,72 |   79,30 |    0,00 |00102| 1814,24 |   96,86 |   | 201203|
110,00 |D 5/8  | 458,00 |3,60 |26,60 |24,50 | 8,70 |42-2| 70,90 | 8,40 |  6,00 |1,38 |0|0,83 |   12,56 |   79,90 |    0,00 |00110| 1985,03 |  106,01 |   | 201203|
126,00 |D 5/8  | 458,00 |3,70 |26,40 |25,40 | 9,00 |52-1| 68,20 | 8,30 |  7,00 |1,45 |0|0,83 |   12,12 |   80,40 |    0,00 |00126| 2056,78 |  107,94 |   | 201203|
 24,00 |D 5/8  | 458,00 |4,00 |26,30 |24,20 | 8,30 |41-1| 74,80 | 8,30 |  3,00 |0,25 |0|0,84 |   11,85 |   80,70 |    0,00 |00024| 1947,52 |  107,20 |   | 201203|
 30,00 |D 5/8  | 458,00 |4,00 |25,80 |28,10 | 9,10 |31-4| 74,30 | 9,40 |  3,00 |0,29 |0|0,83 |   12,57 |   80,30 |    0,00 |00030| 2107,23 |  115,30 |   | 201203|
 37,00 |D 5/8  | 458,00 |3,70 |26,90 |24,10 | 8,80 |31-3| 75,70 | 8,70 |  3,00 |0,29 |0|0,83 |   12,28 |   80,00 |    0,00 |00037| 1966,51 |  108,10 |   | 201203|
 36,00 |D 5/8  | 458,00 |3,90 |26,10 |25,80 | 9,00 |31-4| 74,90 | 8,50 |  3,00 |0,51 |0|0,83 |   11,52 |   81,10 |    0,00 |00036| 2092,99 |  114,37 |   | 201203|
 44,00 |D 5/8  | 458,00 |4,00 |26,40 |27,40 | 8,60 |31-4| 75,10 | 8,90 |  3,00 |0,42 |0|0,83 |   12,67 |   79,90 |    0,00 |00044| 2057,37 |  113,03 |   | 201203|
 49,00 |D 5/8  | 458,00 |3,90 |26,50 |26,00 | 8,10 |31-2| 75,60 | 8,50 |  4,00 |0,53 |0|0,84 |   10,18 |   82,20 |    0,00 |00049| 2239,45 |  121,46 |   | 201203|
 49,00 |D 5/8  | 458,00 |4,00 |26,80 |26,50 | 8,60 |31-4| 74,60 | 8,30 |  3,00 |0,28 |0|0,84 |   10,95 |   81,30 |    0,00 |00049| 2170,49 |  117,59 |   | 201203|
 60,00 |D 5/8  | 458,00 |4,00 |26,10 |25,60 | 8,60 |41-3| 74,10 | 8,20 |  4,00 |0,68 |0|0,84 |   11,74 |   80,90 |    0,00 |00060| 2039,14 |  111,38 |   | 201203|
 30,00 |D 5/8  | 458,00 |4,20 |27,00 |27,10 | 8,90 |42-2| 70,40 | 8,30 |  3,00 |0,37 |0|0,84 |    9,28 |   82,90 |    0,00 |00030| 2337,94 |  122,81 |   | 201203|
 23,00 |D 5/8  | 458,00 |3,90 |26,90 |25,30 | 7,90 |41-1| 76,00 | 8,00 |  3,00 |0,17 |0|0,84 |   11,11 |   81,10 |    0,00 |00023| 2106,75 |  115,19 |   | 201203|
```

Línea 108, columna 305

Figura 7.4 Plano Original

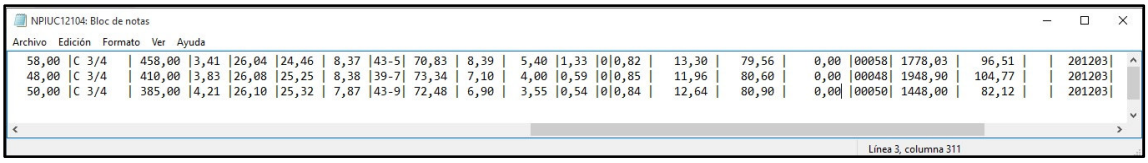


Figura 7.5 Plano Mejorado

7.7. IDENTIFICAR LAS FUENTES DE VALOR Y DESPERDICIO

A lo largo de este trabajo quedó en claro que deberíamos entender por valor a todos los momentos de acción o pensamientos para crear el producto que el cliente está dispuesto a pagar. En otras palabras, el valor se define en los ojos del cliente externo. Se genera en los momentos en los cuales el cliente estaría dispuesto a meter la mano en su bolsillo y pagar por el trabajo.

Por otro lado, podríamos decir que los desperdicios son todos los otros momentos o acciones que no agregan valor, pero consumen recursos. Se pudo comprobar que los residuos provienen de los trabajadores sobrecargados, cuellos de botella, esperas, transferencias, pensamientos poco realistas y la dispersión de la información, entre muchos otros.

Una especie de análisis utilizado en el pensamiento Lean es la estimación de todos los momentos generadores de residuos o de valor. A partir de la cronología obtenida se puede resumir el tiempo de valor y el tiempo de espera, para luego calcular la relación de valor generado. Mediante el estudio de la bibliografía se tuvo conocimiento de que normalmente en una organización de desarrollo no se encuentra una relación de valor superior al 7%. En otras palabras, el 93% (o más) del tiempo aplicado al desarrollo era desperdiciado.

Luego de haber estudiado los conceptos de valor y residuos y haber identificado sus fuentes, se llegó a una diferencia notable en la mejora Lean. Otros sistemas se centran en el perfeccionamiento de las acciones de valor existentes, como por ejemplo, la mejora de las habilidades de diseño, lo que es un objetivo valioso sin lugar a dudas.

Sin embargo, dado que normalmente hay unos pocos momentos de valor añadido en la línea de tiempo (quizás un 5%), entonces la mejora planteada no equivale a mucho. Pero fue imposible pasar por alto el hecho de que al tener una gran cantidad de tiempo desperdiciado en el proceso, había grandes oportunidades de mejorar la relación de valor mediante la eliminación de desperdicios.

Un ejemplo común de desperdicio encontrado en el desarrollo del producto fueron los residuos de sobreproducción, que en términos más simples sería la creación de soluciones o características no deseadas por el cliente. Se llegó a la conclusión de que no tiene mucho sentido enfocarse en medir y mejorar la eficiencia de la ingeniería en un pequeño porcentaje si hay una enorme cantidad de desperdicio en forma de funciones no utilizadas, todas ellas generadas por malas decisiones en todas las etapas anteriores del proceso de desarrollo.

Otro ejemplo inmediato de residuos que podríamos citar es el generado por las esperas o demoras. Los clientes no pagan por ello y es muy fácil perder la noción de todo el desperdicio que se genera por la espera de una aclaración, una aprobación o que otro equipo o integrante termine su parte.

En función de todas las observaciones realizadas en el desarrollo del caso de estudio, se pudieron identificar las siguientes acciones que no agregarían ningún tipo de valor y por lo tanto son desperdicio:

Sobreproducción de soluciones o características, se notó que en el afán de entregar valor lo más rápido posible, se tendía a comenzar a tratar elementos de etapas siguientes sin terminar los de la etapa actual, retrasando todo el trabajo. Se buscó evitar la clara tendencia a desarrollar funciones o servicios que los usuarios finales de sistema realmente no necesitaban o no querían. También se redujo al mínimo la documentación extensa y poco utilizable, concentrándose más en los elementos que podían ser implementados rápidamente, evitando sobre todo la duplicación de esta información, lo que consume tiempo y recursos.
Esperas y retrasos para aclarar cuestiones y realizar documentación. Se notó que al incurrir en estas esperas, todo el proyecto se atrasaba. Esto fue aún más evidente en los tiempos muertos generados al esperar que uno de los integrantes del grupo finalizara su parte del trabajo.
Cualquier tipo de transferencia, transporte y movimiento. En el contexto de equipos de desarrollo con varios integrantes, podríamos citar ejemplos como dar especificaciones de un analista a un ingeniero o que una característica sea probada por un equipo que no fue el encargado de su creación. En el caso del desarrollo de este trabajo, este punto fue abordado bajo la simple pero estricta regla de que cada integrante debía terminar su tarea asignada. Obviamente la colaboración mutua se dio constantemente, pero la responsabilidad de finalizar el trabajo siempre fue de la persona a la que se le había encomendado. Esto potenció el compromiso de cada integrante en cumplir con sus tareas y evitar de esta manera todos los retrasos generados por las transferencias.
Procesamiento adicional, reaprendizaje y reinventar características sobre las cuales ya se trabajó y se dio por finalizadas. Esto hace especial énfasis en analizar bien los requisitos y necesidades del cliente. Cualquier tipo de modificación en algo que se consideraba finalizado es netamente desperdicio.
Trabajo parcialmente hecho, diseños documentados pero no construidos o elementos construidos pero no integrados o probados. Consideramos que estos son los principales exponentes de lo que consideramos desperdicio. Se invierte esfuerzo y recursos en elementos que nunca generan utilidad alguna. Esto se pudo ver más sustancialmente al encontrarse cuellos de botella en el desarrollo. Se trabajó en características que terminaron siendo muy dificultosas de implementar y que por ende quedaron descartadas, generando importante desperdicio de tiempo.
Defectos, pruebas y correcciones después de la creación del producto. Encontrar y eliminar los defectos no es una acción que genere valor, ya que en esencia consiste en arreglar algo que debería haberse hecho bien la primera vez. Evidentemente, este punto depende mucho de las habilidades de cada integrante del equipo, pero de todas maneras se trabajó con un claro enfoque de “cero defectos”.
Uno de los desperdicios más difíciles de identificar fue subestimar o no darse cuenta de las habilidades posibles y variadas de las personas. Se intentó dar lugar a la visión, ideas y sugerencias de los integrantes del equipo. Con esto se trató de que se tuviera constantemente la oportunidad aprender a descubrir los desperdicios y prevenirlos o solucionarlos de acuerdo al caso.

Pérdida de información o dispersión de la misma. Se evitó difundir datos a través de muchos documentos por separado ya que se desperdicia tiempo en interpretarlos y unir las diferentes partes. Un punto muy significativo en este desperdicio fueron las barreras de comunicación. La metodología apunta precisamente al trabajo en un mismo recinto para facilitar y afianzar la comunicación y colaboración de los miembros del equipo, pero en el caso de los investigadores que realizaron este trabajo, fue muy difícil. Al estar radicados en distintas provincias, la comunicación se valía de todos los elementos posibles (correos, videollamadas, programas de acceso remoto), pero inevitablemente se produjo pérdida de información. Se hizo evidente la importancia de trabajar de manera conjunta en un mismo espacio físico y se sufrieron los problemas consecuentes de no hacerlo.
Por último, se intentaron erradicar las expresiones de deseo y pensamientos poco realistas. El excesivo optimismo puede generar malas decisiones o retrasos. Se apuntó constantemente a decidir lo más tarde y con la mayor cantidad de información posible, de manera tal de evitar cualquier tipo de subjetividad en la planificación y ejecución de las tareas. Pese a todo esto, frases como “estamos retrasados, pero lo resolveremos más tarde” aparecieron y dieron claras muestras de los desperdicios que pueden generar, sobre todo de tiempo.

El enfoque adoptado de entregar valor a través de la reducción de residuos orientó el trabajo hacia una organización Lean. Se observó que esta estrategia de mejora es sustractiva en lugar de aditiva. Por ejemplo, en lugar de hacernos la pregunta “¿Qué podemos hacer o implementar para aumentar la productividad?”, la interrogante planteada fue “¿Qué podemos quitar o dejar de hacer?”. En el transcurso de esta investigación se pudo descubrir que esto es un cambio de mentalidad respecto al aseguramiento de la calidad en grandes organizaciones (incluida la que es ámbito del presente trabajo), las cuales se centran en el cumplimiento de checklists y la suma de actividades de “mejora”.

7.8. FLUJO

Este concepto sugiere crear un caudal de valor sin demoras hacia el cliente. Es muy fácil generar confusión sobre esto, ya que podría pensarse que una solicitud de un cliente que está en espera de ser aprobada, analizada, implementada, reprocesada o probada es un buen ejemplo de flujo, pero no lo es. El concepto apunta más bien a cómo se crea valor (en productos, software, información, decisiones, servicios) que fluye inmediatamente al cliente. Está relacionado con el seguimiento y dirección hacia la meta de entregar rápidamente valor. El flujo es un reto de perfección; cero residuos en el sistema y entrega inmediata, continua y fluida de valor constituyen desafíos profundos probablemente nunca alcanzables, pero la idea es siempre apuntar en esa dirección y moverse hacia el flujo.
En base al estudio de la bibliografía, se determinó que avanzar hacia el flujo está asociado a la teoría de colas aplicada, sistemas pull y más. Mediante la comprensión de estos conceptos se pudo apuntar hacia el movimiento con el flujo a través de paquetes de trabajo más pequeños, colas de menor tamaño y reducción de la variabilidad.

7.9. BENEFICIOS DE REDUCIR EL TAMAÑO DEL LOTE Y TIEMPO DE CICLO

En el desarrollo del caso de estudio, se decidió indagar sobre la utilidad de trabajar en lotes de pequeño tamaño y ciclos cortos. Luego de un proceso de análisis y modificación de conductas y enfoques en el desarrollo, se llegó a los siguientes resultados:

Se percibió una reducción global en el tiempo del ciclo de liberación, obtenido a partir de la erradicación de colas y mediante la aplicación de gestión de colas, obteniendo así que muchos ciclos sean más cortos.
Eliminación de retraso por lotes, donde una parte de una solución es innecesariamente retenida debido a que se mueve a través del sistema conectada a un lote mayor de otras soluciones. La eliminación de esto proporcionó otro grado de libertad para el negocio, consolidando el envío de un producto más pequeño mucho antes con las soluciones de más alta prioridad.
Aplicación de la *Metáfora del Lago y las Rocas*: aplicada en la filosofía Lean, la misma toma la profundidad del agua como una representación del nivel de inventario, tamaño del lote o tiempo del ciclo. Cuando el nivel del agua es alto (lotes grandes, mucho inventario o ciclos largos), muchas rocas están ocultas. Estas rocas representan debilidades. En el caso del presente trabajo, se consideró un ciclo de liberación secuencial ocho meses con una gran transferencia de lotes, pruebas e integración ineficientes y escasa colaboración. Estos factores permanecían ocultos debajo de la superficie de un ciclo largo y lotes grandes. Luego se determinó la entrega de pequeños conjuntos de soluciones potencialmente entregables cada dos o tres semanas y de repente todas las prácticas ineficaces se volvieron obvias.
Dicho de otra manera, el costo de transacción del ciclo de proceso antiguo se convierte en inaceptable. Esto se convierte en una fuerza para la mejora, ya que volver a experimentar en cada ciclo corto se torna tedioso y de hecho, puede ser simplemente imposible lograr los objetivos del bucle utilizando las viejas e ineficientes prácticas.

7.10. APRENDIZAJE MÁS VALIOSO Y MENOS COSTOSO

Es bien sabido que no todo conocimiento o información nueva posee valor; lo ideal es crear nueva información económicamente útil. Esto es un reto porque es un proceso de descubrimiento, a veces lográndolo y otras no.
En el desarrollo de este trabajo se utilizó una estrategia Lean general, basada en una simple idea de la teoría de información, la cual determina que se debe aumentar el valor de la información creada y reducir el costo de la creación de conocimiento.
Para trabajar en la obtención de información de más alto valor y menor costo, se utilizaron varias ideas de ayuda. Por ejemplo:

Enfocarse en cosas inciertas: si se elige implementar y probar cosas poco claras de manera temprana, el valor de la retroalimentación es alto precisamente porque los resultados son menos predecibles (las cosas predecibles no nos enseñan mucho).
Centrarse en las primeras pruebas y la retroalimentación: la información tiene un costo real de demora, lo cual es una razón de porqué probar solo una vez al final de un ciclo

secuencial largo (acción motivada por la creencia equivocada de que así se bajarán los costos de prueba). Puede ser muy costoso descubrir durante las pruebas de rendimiento, después de varios meses de desarrollo, que una decisión de arquitectura clave era inadecuada. Después de analizar las diversas fuentes de información, se llegó a la conclusión de que en el desarrollo Lean los ciclos cortos con retroalimentación temprana son críticos y que mediante la implementación de las cosas menos predecibles al principio (en ciclos cortos que incluyan pruebas) el costo de demora se reduce.
Apuntar a la automatización de pruebas a gran escala para aprender acerca de los defectos y el comportamiento. El costo de re ejecutar frecuentemente las pruebas automatizadas suele ser insignificante en comparación con la valiosa retroalimentación temprana.
Enfocarse en la integración frecuente o continua para aprender acerca de los defectos y la falta de sincronización. Luego, mediante la integración frecuente en lotes pequeños, el equipo podría reducir el costo general promedio por el efecto “del lago y las rocas”.
Tener en cuenta y aplicar las enseñanzas de expertos y la difusión del conocimiento para reducir el costo de redescubrimiento.

7.11. CADENCIA

Trabajar en ritmos regulares o cadencia es un principio Lean, tanto en la producción como en el desarrollo. En el primer ámbito, es llamada *tiempo de procesamiento*, mientras que en el desarrollo Lean se utiliza el término *cadencia*. Es un principio de gran alcance en el desarrollo de productos Lean, por lo que el tema se examinó con cierto detalle.
Pudimos notar que existe algo muy básico y humano respecto a la cadencia: las personas aprecian o desean ritmos en sus vidas o trabajos y buscan tener rituales dentro de estos ritmos. La mayoría de nosotros trabajamos en base a una cadencia de semanas de siete días o tenemos el ritual de alguna reunión semanal en un día establecido. Simplemente, se pudo observar que la cadencia en el trabajo mejora la predictibilidad, planificación y coordinación. En un nivel más profundo, podríamos decir que refleja los ritmos por los cuales vivimos nuestras vidas.
Se profundizó sobre un método popular para mejorar la cadencia llamado *timeboxing*, que podría definirse como un ciclo de tiempo de trabajo de desarrollo fijo y usualmente corto. Se trató de lograr que el equipo entregue o demuestre algo al final de la duración fija, idealmente algo pequeño y bien hecho en vez de funciones grandes y parcialmente terminadas. La duración del ciclo no puede cambiar, pero el alcance del trabajo puede variar para adaptarse al límite de tiempo. *Timeboxing* no es una panacea para todos los problemas referidos al conocimiento en el trabajo, pero se vieron algunas ventajas:

- Timeboxing impone la cadencia.
- El trabajo de desarrollo está a menudo poco limitado (o débilmente acotado). Cuando el equipo sabe que el plazo del timeboxing termina en una fecha determinada, limita el trabajo difuso y aumenta el foco. Por lo tanto, el método limita el desplazamiento de los plazos y la sobre regulación y aumenta la concentración.
- Reduce la parálisis del análisis.
- Timeboxing es un contrapeso al mal hábito de dejar las tareas para último momento.

- Si se debe entregar algo bien hecho en exactamente dos semanas, los residuos y la ineficacia de las formas de trabajo actuales se vuelven dolorosamente evidentes. El método crea un cambio hacia la mejora del efecto “del lago y las rocas”.
- Simplifica la programación.
- Las personas son probablemente más sensibles a la variación del tiempo que a la variación del alcance: la frase “era tarde” es recordada con más fuerza que “tenía menos de lo que quería”. Timeboxing reduce la erosión de la confianza que siente la gente al escuchar frases como “tal vez en una semana más todo esté hecho”.

7.12. ¿LAS ENSEÑANZAS DE LA PRODUCCION LEAN PUEDEN AYUDAR AL DESARROLLO?

El desarrollo de nuevos productos o la investigación y desarrollo no son tareas que puedan ser consideradas predecibles y repetitivas como un proceso de fabricación. La suposición de que son similares es una de las causas del mal uso de las prácticas de gestión de “economía de escala” de la manufactura de principios de 1900, aplicadas al desarrollo (por ejemplo, desarrollo secuencial y grandes transferencias de especificaciones en lotes).
Sin embargo, algunos de los principios e ideas aplicadas en la producción Lean, incluyendo ciclos cortos, lotes pequeños, gestión visual y la teoría de colas, se aplican con éxito en el desarrollo de software. La razón de esto es que la producción Lean moderna es diferente; los lotes pequeños, colas y tiempos de ciclo reflejan en parte la visión de la teoría de colas (entre otras fuentes de conocimiento), una disciplina creada para el comportamiento variable en redes que se relaciona mucho más con el desarrollo de productos que con la fabricación tradicional.
Una ironía presente en algunas organizaciones de productos es que los ingenieros de fabricación revolucionaron y adoptaron la producción Lean, alejándose de las “economías de escala” y apuntando hacia el flujo y la flexibilidad en pequeños lotes sin desperdicios. Pero estas lecciones que se ajustan bien al desarrollo de nuevos productos no se utilizan por la gestión del desarrollo, que continúa aplicando prácticas que se encuentran en la antigua gestión de fabricación de economías de escala.
Dicho todo esto, debemos resaltar y advertir que el desarrollo de productos de software no es lo mismo que la industria manufacturera y por lo tanto las analogías entre estos dos dominios son frágiles. A diferencia de la producción, el desarrollo de software es (y debe ser) lleno de descubrimiento, cambio e incertidumbre. Cierta variabilidad es normal y deseable en el desarrollo; de lo contrario, nada nuevo se hace. Por lo tanto, el pensamiento Lean incluye prácticas únicas para el desarrollo de nuevos productos de software.
Mientras más se investigó sobre el pensamiento Lean, fue fácil ver que se trata de un sistema amplio que abarca todos los grupos y funciones de una organización, incluyendo el desarrollo de productos, ventas, producción, servicios y recursos humanos.
La filosofía Lean es mucho más que herramientas como la gestión visual, la administración de colas o solamente la eliminación de residuos. Como puede verse en Toyota, es un sistema empresarial que descansa sobre la base de maestros del pensamiento Lean, asentado sobre pilares como el respeto por las personas y la mejora continua. Su introducción con éxito en empresas puede tomar años y requiere educación generalizada y

entrenamiento, pero muchas de sus prácticas son relativamente fáciles de aplicar en el desarrollo de software.

7.13. RESULTADOS DESPRENDIDOS DEL PROCESO DE DESARROLLO

El proceso de desarrollo de la aplicación que sirvió de caso de estudio, principalmente permitió aplicar muchos de los conocimientos plasmados en el marco teórico.
Después de haber visto los distintos factores que pueden llevar a un proyecto de desarrollo de software hacia el éxito o el fracaso y la importancia del enfoque Ágil y de la filosofía Lean para cualquier equipo de desarrollo, se desprendieron como resultado dos preguntas que consideramos de gran relevancia:

¿Qué prácticas serían consideradas de mayor importancia para ser implementadas en el desarrollo de software?
¿Qué se puede hacer en una organización para mejorar la productividad y aumentar el valor de negocio en la industria del desarrollo de software?

7.13.1. Mejores Prácticas para el Desarrollo de Software

Cuando se indagó acerca de que se consideraría mejor de aplicar al desarrollar una aplicación, se pudo mencionar una buena variedad de prácticas; algunas respaldadas por su utilización y otras fundamentadas desde la teoría. Dentro de las mismas podemos mencionar:

Determinar claramente los requisitos y tener una buena comprensión de los mismos antes de empezar a escribir código.
Crear revisiones conjuntas para que todos los involucrados en el proceso tengan una comprensión común de las cuestiones pertinentes al trabajo.
Apuntar a que el equipo de trabajo se organice por sí mismo en lugar de manejarse en función de un cronograma de trabajo y el control de su cumplimiento.
Fomentar y apoyar la comunicación y el debate entre los miembros del equipo, pero siempre con la clara meta de buscar la mejor solución para las cuestiones que puedan surgir.
Creación de acuerdos y procesos claros para todos los involucrados en el trabajo.
Dar la importancia apropiada a la longitud de las iteraciones y trabajar con los mismos límites para que todos los integrantes aseguren una cooperación eficaz. Un método efectivo en este punto es timeboxing.
Contar con un equipo de personas coherente y evitar cambiar miembros durante el transcurso del proyecto. También se debe asegurar una retroalimentación estrecha entre todos los integrantes.
Tener bien en claro el significado y la manera de llevar un buen rumbo, teniendo como objetivo la mejora continua.
Los participantes del trabajo deben obtener los conocimientos y habilidades adecuadas, promoviendo su intercambio y transferencia. Esto facilita la obtención de retroalimentación entre los miembros del equipo.
Entregar funciones al cliente lo más pronto posible, lo que generará feedback más rápido.

7.13.2. Mejora de la Productividad

En función a este punto, podríamos considerar que es importante contar con experiencia en la industria del desarrollo de software y poder así brindar una opinión fundamentada en base a hechos. Sin embargo, el estudio de la bibliografía y el desarrollo del caso de estudio crearon una cierta sensación sobre qué “hacer” y “no hacer” en lo referido a la mejora de la productividad. En una observación general, podríamos destacar los siguientes puntos:

Medir la productividad y encontrar las causas fundamentales de los posibles problemas y solucionarlas.
Eliminar las actividades de trabajo inútiles y enfocarse en las de alto valor.
Asignar las tareas adecuadas a las personas competentes, lo que implica permitir que las personas trabajen en lo que son mejores.
Motivar a los miembros del equipo a comprometerse con el trabajo que están realizando.
Asegurar que los miembros del equipo tengan contacto fluido con los clientes o futuros usuarios del sistema.
Utilizar prácticas agiles para mejorar la eficiencia del trabajo en equipo y mejorar su velocidad de trabajo en forma continua.
Centrarse en la formación de las personas y construir las competencias, ayudándolas a aprender y mejorar.
Preservar los procedimientos y acuerdos existentes que han demostrado ser útiles en el pasado. Lean definitivamente no significa que no habrá procedimientos o procesos.
Tratar de hacer las cosas bien la primera vez.
Enfocarse en la mejora de la productividad del equipo en lugar de la de los individuos específicos.
Considerar la gestión del cambio, pedir retroalimentación y tratar de ser abiertos.
Criticar y combinar métodos de desarrollo.
Asegurarse de que el equipo de desarrollo comprende el valor de negocio de lo que se entrega.
Aplicar técnicas para entender lo que el cliente realmente quiere y acortar la retroalimentación tanto como sea posible.
Incluir la experiencia del usuario en la evaluación de la eficacia del producto que se está haciendo y de lo que ya se ha producido.
No asumir que todos los miembros del equipo estarán motivados por la misma actividad. Se debe fomentar el intercambio de conocimientos y llegar a tener equipos multifuncionales reales.
Las asignaciones basadas en el tiempo con algún beneficio objetivo ligado al rendimiento aumentarán la productividad de los desarrolladores y el valor de negocio.

En un nivel más amplio, es interesante apuntar hacia la productividad en función al estilo de gestión del proyecto. Se pudo inferir que un problema real de las organizaciones que experimentaron fracasos de proyectos son administradores que piensan que lo saben todo. Esta falta de comprensión por parte de la gestión podría considerarse uno de los mayores impedimentos para la mejora de la industria en su conjunto, ya que tiene a todos tirando en direcciones diferentes. Unos pocos minutos de trabajo imprudente por parte de un gerente puede deshacer semanas de trabajo duro en busca de mejoras por parte del equipo.

Desde el punto de vista del respeto por las personas, se pudo determinar que los factores humanos son la clave para motivar y comprometer a los integrantes del equipo. Es muy importante tener conocimientos en el estudio de la conducta humana y ser emocionalmente inteligentes en beneficio del trabajo en equipo, para lograr de esta manera altos resultados que conducen a un mayor valor de negocio. Las personas deben ser motivadas e influenciadas para lograr resultados superiores, aumentando su satisfacción y desempeño.
A partir del estudio de las fuentes que conforman el marco teórico y el desarrollo del caso de estudio, se hizo evidente la existencia de una conexión entre la productividad y la satisfacción del trabajo. Sin embargo, cuando se trata de los beneficios que el uso de la metodología Lean aporta, se tiende a ver el impacto en el proceso de trabajo más que en los equipos. En nuestra opinión, esto podría ser debido a la naturaleza del dominio del desarrollo de software, el cual a menudo tiende a evaluar las habilidades técnicas y de ingeniería de los desarrolladores y la automatización de los procesos de trabajo, poniendo menos énfasis en las habilidades y emociones de la gente, lo que consecuentemente resta importancia a los aspectos blandos de la gestión que podrían mejorar la productividad del equipo.

7.14. DESARROLLO DE UN SISTEMA "MAGRO"

Durante todo el proceso que implicó la realización de este trabajo, se intentaron poner en aplicación la mayor cantidad de principios y prácticas que permitieran las relativas limitaciones con las que se contaba. Dentro de las mismas podemos mencionar el tamaño del equipo, ya que muchos de los conceptos plasmados se ajustan a grupos de desarrollo de más de 5 miembros y divididos es áreas de especialización. En este punto, todas las conclusiones obtenidas se basan en la teoría y deducciones en función de casos de estudio más que de su aplicación real en el ámbito de este trabajo.
Otra limitación en la aplicación de muchos conceptos teóricos fue la idoneidad técnica de los autores de este trabajo. Muchas prácticas propias de la metodología requerían gran experiencia en el manejo de herramientas específicas de programación, pero el objetivo de este trabajo no es mostrar su funcionamiento, sino fundamentar los principios que sustentan su aplicación.
Una vez definido con certeza el alcance y las limitaciones, el proceso de desarrollo se llevó a cabo tomando como premisas la eliminación de todo lo que pudiéramos considerar desperdicio sin aporte de valor para el usuario.
Se estudiaron a fondo los requisitos iniciales y a medida que el desarrollo avanzó, fueron siendo depurados. Se eliminaron funciones que realmente eran innecesarias y se ahorró esfuerzo al simplificar al máximo la interfaz de usuario.
Todo esto decantó en la obtención de una herramienta simple y rápida, con las funciones estrictamente necesarias para el usuario final y con una interfaz austera, pero no por esto menos eficaz y eficiente.
En todo momento se intentó seguir la filosofía Lean y como principal resultado de esto, se obtuvo una herramienta optima desde el punto de vista de los recursos aplicados a su construcción y que se ajusta estrictamente a las necesidades reales del usuario.

8. CONCLUSIONES

La filosofía del pensamiento Lean propone metodologías sencillas para mejorar el proceso de producción, como así también en el rendimiento y calidad de los recursos productivos en los que se aplica, lo que permite detectar e identificar algunas fuentes de perfeccionamiento en el trabajo mediante la eliminación y reducción del desperdicio.

Con el proyecto de software propuesto intentamos alcanzar los principios expuestos(sin abarcar la totalidad de la metodología ya que está ideada para proyectos grandes), a lo largo de este trabajo con el fin de obtener resultados lo más cercano posible a lo requerido por la empresa, incorporando métodos de calidad y promoviendo que el personal de trabajo esté motivado a mejorar continuamente la forma de llevarlo a cabo para que incida en la calidad sistemática del desarrollo del producto final.

El objetivo fundamental de esta tesis fue abordar la problemática propuesta por la Empresa COTECA S.A. para poder aportar una solución en las tareas de hilado a partir de la Metodología de Desarrollo Ágil conocida como **Lean Software Development** o **Desarrollo de Software Ajustado**, el cual basa sus principios sobre su mismo nombre aplicado a los sistemas de producción, que también se lo denominó como “producción justo a tiempo”.

Esta investigación es una contribución a la mejora de la calidad del trabajo que tiene como premisa la adaptación de un modelo de desarrollo para la construcción de un software de calidad basado en estándares establecidos por manufactura esbelta (*Lean Manufacturing*), ésta se compuso desde la evolución de los principios de calidad desarrollados por la empresa Toyota donde se desplegaron un conjunto de técnicas de gestión, las cuales no solo permiten desarrollar productos sino procesos de calidad, a través de herramientas, tales como: el mapa de la cadena de valor, la tarjeta visual y otras instrumentos que permiten consolidar más eficientemente el proceso de producción.

Ahora bien esta filosofía en el ámbito de la Ingeniería del Software, lo denominamos desarrollo de software esbelto, el cual fue de gran utilidad para mejorar la calidad del producto final y efectuar el desarrollo en forma ágil, de este modo se ha estudiado e investigado la manera de aplicar la manufactura esbelta al desarrollo de software. Los componentes del software están altamente personalizados y fueron construidos bajo la demanda de la empresa, empleando las herramientas propias de la metodología Ágil Lean donde se destacó entre sus características lo siguiente: se *eliminaron las fuentes de desperdicio*, se trabajó con *documentación mínima requerida*, se *detectaron* las *causas raíz del error*, se *programó por pares*, se efectuaron *revisiones de diseño*, se *implementaron prototipos*, se *establecieron estándares de codificación* y finalmente se automatizaron las disciplinas básicas de programación.

Los principios ágiles promueven que los equipos de trabajo sean auto-organizados, que además estos deliberan permanentemente sobre como los elementos de trabajo necesitan mejorar. Como ya se sabe, el desarrollo del trabajo se realizó en el Sector Materia Prima de COTECA S.A., como campo de estudio para confeccionar el mismo en los plazos propuestos, también para cumplir con la cantidad de características informadas para que el

software pueda funcionar y realizar los cambios en los requerimientos solicitados y manifestados por la misma entidad fabril, lo que permitió una reducción en el ciclo de las tareas que se realizan en dicho Sector, la cual se dedica principalmente a la generación de calles de fardos de algodón en relación a un conjunto de parámetros de calidad asignados para el resultado final.

Como ventaja principal al investigar y desarrollar sobre de la Metodología fue que se pudo contribuir a mejorar, integrar y flexibilizar los procesos de producción, la cual se puede aplicar y beneficiar para cualquier ámbito de producción.

Considerando las implementaciones de la Metodología Lean en sus principios y prácticas podemos definir como conclusión obtenida al desarrollar el presente trabajo lo siguiente:
La ejecución del software incidió positivamente en la automatización del proceso de hilado, con el propósito de que fuera capaz de simular y optimizar con los requisitos que se definieron en base a las necesidades detectadas y/o demandadas por la empresa, logrando así solucionar un trabajo que se realizaba de manera artesanal.

Se comprobó que implementando la Metodología de Desarrollo Ágil en la sistematización del trabajo el cual se ha diseñado mediante un lenguaje unificado aportando una visión detallada y explicita de los requisitos definidos, especificando su funcionamiento de acuerdo al estudio realizado. La construcción del sistema propuesto es acorde a las necesidades de la empresa que dio lugar a una arquitectura efectuada en la etapa de diseño la cual implicó la programación y generación de código fuente para la aplicación.

La utilización de las tecnologías basadas en Lean nos proporcionó facilidades mediantes sus principios permitiendo que se erradiquen o disminuyan las ineficiencias detectada, logrando así un incremento en el rendimiento y desempeño de las tareas diarias, lo cual ha permitido beneficiar en la organización y fluidez del proceso de producción, cumpliendo a su vez, con los tiempos y objetivos demandados en el trabajo con eficiencia.

En cuanto a las pruebas automatizadas que se realizaron empleando las herramientas utilizando *pruebas unitarias*, podríamos destacar como unos de los beneficios principales de contar con estos test es que se reutilizan y se puede acumular código indefinidamente, además en el caso de los errores se pueden detectar con anterioridad lo que por sí solo implica un ahorro de tiempo, costos de despliegue y mala imagen.

Podemos agregar además que los marcos de trabajos ágiles aportan mayor flexibilidad, retroalimentación de la calidad, eficacia y rapidez, reduciendo el margen de error y los riesgos del proyecto. Es por esto que se seleccionó tal Metodología como patrón de estudio.
Finalmente se puede concluir que el presente trabajo permitió aplicar los conocimientos adquiridos a lo largo de la carrera como así también incorporar nuevos conceptos que nos serán útiles para desempeñarnos en el ámbito profesional.

9. REFERENCIAS

[1] Royce, “Gestión del desarrollo software de grandes sistemas”.

[2] Shingo, “Estudio de Sistema de producción Toyota”.

[3] DeMarco y Lister, "Peopleware: Productive Projects and Teams"

[4] Royce, “Managing the Development of Large Software Systems”

[5]Johnson, “ROI, It's Your Job”

[6] Schwaber y Beedle, “Agile Software Development with Scrum”

[7] Schwaber y Beedle, Agile Software Development with Scrum

[8] Highsmith, “Adaptive Software Development”

[9] Johnson, “ROI, It's Your Job”.

[10] Palmer and Felsing, “A Practical Guide to Feature-Driven Development”

[11] Battin, Crocker, Kreidler y Subramanian, “Leveraging Resources in Global Software Development”.

[12]Sobek, “Principles That Shape Product Development Systems: A Toyota-Chrysler Comparison”

[13] Ward, Liker, Cristaino y Sobek, “The Second Toyota Paradox: How Delaying Decisions Can Make Better Cars Faster”.

[14 - 15] Clark and Fujimoto, “Product Development Performance”

[16] Kajko-Mattsson, “Taxonomy of Problem Management Activities”.

[17] Boehm, “Industrial Software Metrics Top 10 List”.

[18] Boehm y Papaccio, “Understanding and Controlling Software Costs”, 1465–1466.

[19] Boehm y Basili, “Software Defect Reduction List”.

[20] Boehm y Papaccio, “Understanding and Controlling Software Costs”, 1465–1466.

[21] Coy, “Exploring Uncertainty”.

[22]Zaninotto, “From X Programming to the X Organization”.

[23] Highsmith, “Adaptive Software Development”

[24]The Lean Construction Institute coined the term last responsible moment. Verwww.leanconstruction.org.

[25]Thimbleby, “Delaying Commitment”

[26] Klein, “Sources of Power”, Chapter 3

[27] Idem 26

[28] Miller, “The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information”

[29-30] Womack, Jones y Roos, “The Machine That Changed the World”

[31-32]Ohno, “The Toyota Production System”

[33]Capítulo 5, “EmpowertheTeam.”

[34] Capítulo 2, “Herramienta 4: Iteraciones.”

[35]Extreme Programming Installed, Capítulo 4.

[36]The pivotal importance of a daily meeting is described in Schwaber and Beedle, Agile Software Development with Scrum,

[37] Ohno, “The Toyota Production System”

[38] Cockburn, “Agile Software Development”

[39]Herramienta 20: Testing,” in Capítulo 6, “Build Integrity In”

[40] Goldratt, “Theory of Constraints”, y Goldratt, “The Goal”

[41]Reinertsen, “Managing the Design Factory”

[42-43] Herzberg, “One More Time: How Do You Motivate Employees?”

[44] Imai, Gemba Kaizen

[45] Humphrey, “Winning with Software”

[46]Highsmith, “Agile Software Development Ecosystems”

[47] Humphrey, “Winning with Software”.

[48]Koskela y Howell, “The Underlying Theory of Project Management Is Obsolete”

[49]Capítulo 7, “See the Whole”

[50] Adler, “Time and Motion Regained”

[51] Ohno, “The Toyota Product System”

[52] Collins y Porras, “Built to Last”

[53] Thomas, “Intrinsic Motivation at Work”

[54]Sobek, Ward y Liker, “Toyota's Principles of Set-Based Concurrent Engineering”

[55-56] Curtis, Kransner y Iscoe, “A field Study of the Software Design Process for Large Systems”

[57]www.agilemanifesto.org

[58] Johnson, “GUI Bloopers”

[59] Clark y Fujimoto, “Product Development Performance”

[60]Kajko-Mattsson, “Taxonomy of Problem Management Activities.”

[61-62] Clark and Fujimoto, Product Development Performance

[63]Fowler, “Patterns of Enterprise Application Architecture”

[64]Hohmann, “Beyond Software Architecture”

[65]Petroski, “Design Paradigms”

[66] Norman, “The Design of Everyday Things”

[67]Hohmann, “Beyond Software Architecture”

[68]Fowler, Refactoring

[69] Hunt and Thomas, “The Pragmatic Programmer”

[70]Marick, “When Should a Test Be Automated?”

[71]Forrester, “System Dynamics and the Lessons”

[72]Senge, “The Fifth Discipline”

[73]Goldratt, “The Goal, and Theory of Constraints”

[74]Goldratt, “Necessary But Not Sufficient”

[75]Senge, “The Fifth Discipline”

[76]Ohno, “The Toyota Production System”

[77-78-79-80] Austin, “Measuring and Managing Performance in Organizations”

[81]Demming, “Out of Crisis”

[82-83-84-85] Dyer, “Collaborative Advantage”

[86-87-88-89-90] Clark and Fujimoto, “Product Development Performance”

[91]Dyer, “Collaborative Advantage”

[92] Thompson, “Public Economics and Public Administration”

[93]Dyer, “Collaborative Advantage”

[94]Ripin and Sayles, “Insider Strategies for Outsourcing Information Systems”

[95]Thompson, “Public Economics and Public Administration”

[96]Thompson, “Handbook of Public Administration”

[97]Pitette, “Progressive Acquisition and the RUP: Comparing and Combining Iterative Processes for Acquisition and Software Development”

[98]Hohmann, “Beyond Software Architecture”

[99] Johnson, “ROI, It's Your Job”

[100] Boehm y Papaccio, “Understanding and Controlling Software Costs”

10. BIBLIOGRAFIA

Lean Software Development in Action [Autores: **Janes** , Andrea, **Succi** , Giancarlo]

Agile Software Requirements: Lean Requirements Practices for Teams, Programs and the Enterprise (Agile Software Development Series) [Autor: Dean Leffingwell]

The Art of Lean Software Development [Autores: Curt Hibbs (Author), Steve Jewett (Author), Mike Sullivan (Author)]

Lean-Agile Software Development: Achieving Enterprise Agility [Autores: Alan Shalloway, Guy Beaver, James R. Trott]

Agile Software Development, Principles, Patterns, and Practices [Autor: Robert C. Martin, 2002]

Métodos Heterodoxos en Desarrollo de Software [Autor: Carlos Reynoso – Revisión técnica de Nicolás Kicillof – Universidad de Buenos Aires]

Book Review: Leading Lean Software Development [Autor: Fabio Cevasco, Febrero 2013]

Lean Manufacturing tools aplied to equality software development [Autores: MacringerOmaña y José TomásCadenas, Junio 2011]

Being Agile with Lean Software Development [Autor: Jason Tee]

Agile Modeling[Ambler-2002].

Lean principles, learning, and knowledge work: Evidence from a software services provider [Autores: Bradley R. Staatsa, David James Brunnerb, David M. Uptonc - 2011]

Metodologías Ágiles en el Desarrollo de Software [Autores: Patricio Letelier Torres
Emilio A. Sánchez López - 12 de Noviembre de 2003]

Lean, Agile [Autor: David Harvey, Enero 2004]

Lean Software Development [Autores: Dasari. Ravi Kumar, Junio 2005]

Lean Thinking - Banish Waste and Create Wealth in Your Corporation, Revised and Updated [Autores: James P. Womack y Daniel T. Jones – Junio 2003]

Lean Manufacturing: Tools, Techniques, and How to Use Them [Autor: William M Feld]

The Impact of Agile Principles on Market-Driven Software Product Development [Nina DzamashviliFogelstrom, Tony Gorschek, Mikael SvahnbergyPeo Olsson - 2010]

Agile Methods and CMMI: Compatibility or Conflict? [Autores: Martin Fritzsche, Patrick Keil – 2007]

Toyota Production System: Beyond Large-Scale Production [Autor: TaiichiOhno - 1 Mar. 1988]

Lean Software Development: En (tutorial): 29th International Conference on Software Engineering (ICSE'07),Minneapolis, EEUU [Autor: Poppendieck M. – 2007]

Implementing Lean Software Development: From Concept to Cash [Autor: Mary Poppendieck, Tom Poppendieck – 2007]

Ingeniería del Software – un enfoque práctico – Sexta Edición [Autor: Pressman R. – 2005]

11. SITIOS WEB CONSULTADOS

https://es.wikipedia.org/wiki/Lean_software_development

http://www.scrummanager.net/bok/index.php?title=Lean_Software_Development

http://leansoftwareengineering.com/

http://link.springer.com/chapter/10.1007%2F978-3-642-16416-3_12#page-1

http://www.dosideas.com/noticias/metodologias/410los7principiosdeldesarrollolean.

http://gravitar.biz/bi/metodologias-agiles-intro/

http://www.ecured.cu/Lean_Development

http://albertolacalle.com/hci/lean.htm

http://www.estratecno.es/curso/12/kanban-y-lean-para-la-gestion-gil-de-proyectos-tic

http://www.projectperfect.com.au/downloads/Info/info_lean_development.pdf

12. ANEXOS

12.1. PROCESO PRODUCTIVO EN COTECA S.A.

Figura 12.1 COTECA S.A. – Sector Apertura: se puede apreciar la calle de fardos producto de la gestión técnica.

Figura 12.2 COTECA S.A. – Sector Cardas: aquí se ve la cinta de algodón obtenida luego del proceso de limpieza.

Figura 12.3 COTECA S.A. – Sector Cardas: vemos modelos de cardas más avanzados. Logran mayor limpieza a mayor velocidad.

Figura 12.4 COTECA S.A. – Sector Cardas (otro ángulo).

Figura 12.5 COTECA S.A. – Sector Manuares: en esta parte del proceso productivo se realiza la mezcla de cintas de carda y el estiraje necesario para obtener una cinta más fina.

Figura 12.6 COTECA S.A. – Sector Unilap (unidora de cintas): en esta parte del proceso se unen cintas de manuares y se obtiene una napa que sirve de alimentación a las peinadoras.

Figura 12.7 COTECA S.A. – Sector Peinadoras: en esta área se mezclan y peinan las napas, obteniendo así una cinta con un máximo nivel de limpieza y paralelismo de las fibras de algodón.

Figura 12.8 COTECA S.A. – Sector Mecheras (vista posterior): aquí se realiza el estiraje de la cinta peinada previo al proceso de hilatura propiamente dicho.

Figura 12.9 COTECA S.A. – Sector Mecheras (vista frontal): en la parte superior puede apreciarse la mecha obtenida en esta máquina.

Figura 12.10 COTECA S.A. – Sector Mecheras (vista interior): proceso de formación de la mecha.

Figura 12.11 COTECA S.A. – Sector Continuas: en estas máquinas, llamadas *Continuas de Hilar*, se realiza la hilatura propiamente dicha, emulando el tradicional método de la rueca, se estira y da torsión a la mecha y se van obteniendo *cops* con hilo de las características deseadas.

Figura 12.12 COTECA S.A. – Sector Continuas: en esta vista se puede apreciar la longitud de la máquina, que puede llegar a tener 1.800 puestos de hilatura.

Figura 12.13 COTECA S.A. – Sector Continuas: vista detallada del puesto de hilatura. Pueden verse los *cops*rojos con el hilo obtenido. Los mismos luego serán reunidos y se formará una bobina.

Figura 12.14 COTECA S.A. – Sector Enconadoras: es la parte final del proceso. Aquí se reúnen los *cops*y se van formando bobinas (conos) listo para su comercialización y envío a clientes.

Figura 12.15 COTECA S.A. – Laboratorio de Control de Calidad: vista general del área encargada de asegurar que el hilado producido esté acorde a las necesidades del cliente.

Figura 12.16 COTECA S.A. – Laboratorio de Control de Calidad: RegularímetroUster usado para medir parámetros de calidad del hilado.

Figura 12.17 COTECA S.A. – Laboratorio de Control de Calidad: Dinamómetro Mesdan utilizado para medir la resistencia del hilado.

Figura 12.18 COTECA S.A. – Laboratorio de Control de Calidad: RegularímetroMesdan usado para medir parámetros de calidad del hilado.

Figura 12.19 COTECA S.A. – Depósito: módulos con hilado listos para ser enviados a los clientes.

12.2. ENTREVISTAS

12.2.1. ENTREVISTA AL INGENIERO MECANICO JOSE OMINETTI

SP: ¿Por qué considera que es tan difícil para algunas empresas implementar y alcanzar buenos resultados al implementar nuevos paradigmas de trabajo, sobre todo Just In Time o Lean?

JO: Obviamente cada caso tiene sus particularidades y matices, pero por lo general se puede ver que las medidas de transformación de las empresas tienen dos defectos importantes. El primero es el fracaso para enfatizar un proceso total para la mejora. Esto significa establecer objetivos específicos, medir el progreso hacia la consecución de los objetivos, identificar las causas fundamentales de los problemas en alcanzar los objetivos mediante la utilización de equipos, la formulación de acciones para solucionar esos problemas, medir el progreso con frecuencia, reformular las acciones y comunicar los avances a toda la organización.
Si los mandos de alto rango descuidan alguno de estos elementos, los resultados obtenidos se verán debilitados. Ahora, esto conlleva a la pregunta de ¿por qué más empresas no ven al esfuerzo de transformación como un proceso total? Creo que algunos directivos no saben cómo encarar un proceso total de mejora continua y tienen una mentalidad de buscar soluciones rápidas. Otros carecen de la formación y experiencia en la fabricación u operaciones, incluyendo también técnicas formales de resolución de problemas. Normalmente prefieren resolver los percances de una manera más informal.
En otros casos ni siquiera ven la necesidad de un cambio masivo. Muchas empresas se volvieron demasiado “cómodas” porque tenían el lujo de haber crecido en el pasado.
El segundo error es el fracaso de los altos mandos al no invertir bastante tiempo observando las actividades de mejora continua de los empleados, enseñándoles la forma de aplicar los diversos elementos de mejor continua y hacer los cambios de personal necesarios para optimizar los elementos del proceso total.
Creo que muchos directivos no se involucran con el proceso de mejora porque creen que no tienen el tiempo para involucrarse a ese nivel de detalle. Creen que ese es el trabajo de otra persona en un nivel más bajo de la organización.
Otros son más generalistas, particularmente aquellos que tienen experiencia limitada en fabricación u operaciones. No se sienten cómodos discutiendo operaciones.

SP: Según su criterio y experiencia, ¿Qué debe hacer la dirección para evitar estas fallas?

JO: Tienen que familiarizarse con el proceso total y estar comprometidos en involucrados en los elementos que lo conforman. Esto se puede hacer con la lectura selectiva de libros oartículos y mediante la contratación de un empleado o consultor con experiencia en el tema para actuar como “entrenador personal”. Los altos directivos, a su vez, deben seleccionar y motivar a los gerentes que implementarán Lean o Justo a Tiempo.

SP: ¿Cómo piensa que se tiene que reestructurar la contabilidad para que los beneficios de las mejoras Lean planeadas aparezcan en los balances financieros?

JO: No estoy seguro del grado en el que la contabilidad deba ser restaurada, aunque asumo que se necesitarían más análisis de costos. Lo importante es unir las mejoras específicas en las operaciones (por ejemplo, en nuestro caso, menor tiempo de paros de máquina o husos inactivos) a la reducción esperada de costos específicos y realizar un seguimiento de los mismos de manera mensual, por área y por el total de la planta.
A medida que mejoras las operaciones de fabricación, el equipo de gestión deberá revisar constantemente qué costos pueden reducirse, tomar las medidas necesarias reducirlos y medir las tendencias de los costos mensuales para asegurarse de que las reducciones están ocurriendo en realidad.
A medida que el inventario se reduce a un nivel objetivo donde la producción es aproximadamente igual a la demanda en el tiempo (lo que es típico en cualquier implementación de JIT), la gerencia debe anticipar que puede haber un impacto negativo por única vez en los balances de resultados. Esto se debe a que la reducción de producción puede dar lugar a una menor absorción de costos fijos. Sin embargo, esto es por lo general un impacto a corto plazo y ocurre solo una vez. Por supuesto, todo esto que menciono es en carácter netamente teórico. La empresa aún no ha decidido implementar los conceptos que menciono.

SP: ¿Alguna vez participó de un proyecto de implementación de JIT o Lean?

JO: Efectivamente, lo he hecho. Tuve la oportunidad de formar parte del proyecto de reconversión del proceso productivo en otra firma del rubro hace algunos años. En ese momento, la apertura de importaciones obligó a la gerencia a incurrir en un nuevo paradigma de trabajo para afrontar la fuerte competencia. Los puntos fuertes que se atacaron en ese momento fueron: el traspaso de la producción en lotes hacia células de fabricación, la instauración del flujo continuo, reconvertir la gestión de la demanda y cambiar el sistema de planeamiento.

SP: Desde un punto de vista global, ¿Cómo comenzó el proceso de transformación?

JO: Comenzó con un mapeo general del flujo de valor y hablando con la gente para determinar cuáles eran las oportunidades si la empresa aplicaba los cambios que se estaban planteando.

SP: ¿Cuáles fueron los mayores retos y como los abordaron?

JO: Siempre se encuentran problemas alrededor de todo el proceso y con la gente. Por lo tanto, al final de la jornada, uno de los mayores retos era asegurarse que todos hubieran entendido la visión de lo que estábamos tratando de lograr y de por qué habría valor si se trabajaba duro y se mantenía el rumbo, pese a ser una tarea difícil en la que se estaban exponiendo todos los problemas a medida que se intentaba implementar el sistema.

SP: ¿Qué beneficios se obtuvieron tras la implementación?

JO: En un primer momento se había planeado un proceso de reconversión de tres años, pero terminó durando un poco más de cuatro. Luego de este periodo observamos que la producción y el proceso podían ser controlados de mejor manera, definitivamente el tamaño de stock se redujo en todas sus formas (materias primas, productos en proceso y terminados) y se optimizó el trabajo con los proveedores. También se pudo apreciar una mejora desde el punto de vista de las personas y podríamos considerar que se logró una consolidar el trabajo en equipo (al menos en un primer momento). Todo esto pudo traducirse luego en mermas en los costos de producción.

SP: Para concluir, ¿Considera que TN&Platex en un futuro podría implementar esta filosofía de trabajo?

JO: Definitivamente podría aplicarse en todo lo que sea referido a la mejora de procesos, pero desde el punto de vista de reducción de stocks lo veo tal vez algo más complicado. Nosotros trabajamos con algodón virgen directamente traído desde la desmotadora, lo que dicho en términos más técnicos significa que nuestra materia prima es un commodity y por definición los márgenes de ganancias son bajos. Esto determina que se deba producir en gran cantidad, apuntando a ventas importantes. Evidentemente, la situación económica del país siempre es un factor condicionante y las ventas se mueven de manera vertiginosa. Todo esto desencadena un movimiento de stock poco predecible y al producir a volumen constante, las existencias son variables

12.2.2. ENTREVISTA AL INGENIERO EN COMPUTACION GONZALO HARO

SP: ¿Está familiarizado con el concepto de Desarrollo de Software Lean o alguna vez había oído hablar del mismo?

GH: Realmente no había recibido mucha información respecto a la metodología. A decir verdad, estoy familiarizado con los conceptos y principios que la fundamentan pero en el ámbito de la organización industrial (por Justo a Tiempo), en lo que se refiere al desarrollo de software, realmente no la conocía.

SP: Habiendo ahondado un poco respecto al tema, ¿Qué opinión podría emitir sobre la misma?

GH: Me parece por demás interesante, sobre todo en lo que se refiere al concepto de desperdicio y su eliminación. Cuando uno se involucra en un proyecto de desarrollo, no tiene consciencia normalmente de todo lo que hace que puede ser considerado un desperdicio de tiempo o recursos. Al familiarizarse con el concepto de desperdicio, uno comienza a darse cuenta de que podría trabajar de una manera muchísimo más eficiente y sobre todo, en menor tiempo.

Creemos que por utilizar alguna metodología de desarrollo ágil, ya hemos optimizado la manera de trabajar, pero evidentemente uno pasa por alto muchas cosas y veo que este enfoque las aborda.

SP: Enfocándonos en su experiencia en el desarrollo de software en el ámbito de un equipo de una empresa, ¿Cómo se medía la productividad de las personas y los procesos involucrados a su trabajo?

GH: En la última empresa en la que estuve trabajando, se estaba tratando de implementar la metodología Scrum y tener equipos de desarrollo multifuncionales. De alguna manera es difícil medir la productividad, especialmente en el caso que te menciono, donde se estaba cambiando la metodología y había que empezar de nuevo. En Scrum es necesario comparar la productividad del mismo equipo para diferentes “sprints”. Por lo tanto, había que dejar pasar ciertos periodos de tiempo antes de hacer cualquier comparación (para un equipo específico).

SP: ¿Qué factores de la gestión de proyectos marcan la diferencia en la velocidad y productividad del desarrollo?

GH: Al no tener mucho conocimiento sobre Lean, solo puedo responder desde la perspectiva de Scrum. La función de director del proyecto es llevada a cabo por el “Scrum master”, quien debe apoyar y ayudar al equipo para despejar los impedimentos y que pueda seguir funcionando en la dirección asignada. Las reuniones diarias, entre otras cosas, sirven para ayudar al equipo a aumentar la comunicación entre sí y para que aparezcan los problemas a los que se enfrentan durante el trabajo.
Una cosa muy importante es el seguimiento de los trabajos que se hace al final de cada sprint importante mediante el cual se intentan identificar los problemas que han ocurrido o cosas que uno puede hacer mejor en el siguiente paso. Por lo tanto, ese periodo de tiempo cuando se mira hacia atrás y se trata de resolver los problemas es lo más importante, en Scrum al menos.

SP: ¿El tamaño del equipo de desarrollo puede afectar la productividad?

GH: Desde mi criterio, sí. Creo que no debe haber más de siete u ocho personas en un solo equipo. Si se llegara a necesitar más personas (nunca he estado en un proyecto que lo ameritara), se debería crear otro equipo en su lugar. Esto conduce a mejorar la comunicación y coordinación del equipo.

SP: ¿Los proyectos con ciclos de desarrollo más rápidos, son más productivos?

GH: La mayor ventaja de tener iteraciones cortas es que se puede cambiar de rumbo o comprobar si el desarrollo va en la dirección correcta. Creo que esto es muy importante, porque uno debe asegurarse de que está haciendo lo correcto para tener éxito.

SP: ¿Cómo pueden las metodologías agiles de gestión de proyectos mejorar los procesos y métodos?

GH: No creo que la metodología en particular sea crucial si se sabe bien lo que se necesita para desarrollar. Cuando se sabe qué tipo de producto se va a crear, se puede utilizar hasta la metodología de cascada. Si se trata de un proyecto más complejo en el que no se tienen todas las especificaciones de cliente cuando se arranca, una metodología ágil como Scrum ayudará a llegar al objetivo.
El uso de iteraciones es una cosa fundamental. Tener puntos de referencia en la línea de tiempo permite hacer correcciones según las necesidades del cliente. Hay otra dimensión en el uso de metodologías agiles: tener equipos que se han especializado en un área determinada. Con el tiempo, ese equipo aumentará sus conocimientos y llegará a soluciones que no habría sido posible conseguir de otro modo. En esa dimensión, Scrum y otras metodologías agiles (incluida Lean) son muy buenas. Sin embargo, cabe aclarar que esto va bien si no se cambian los miembros del equipo.

SP: Basado en la experiencia de su última empresa, ¿Cuáles son algunos métodos para migrar todo un equipo hacia una metodología ágil?

GH: No creo que todo el mundo debería operar con la metodología ágil. En realidad se tienen ciertas partes del desarrollo (por ejemplo, aquellas que se centran en las pruebas de rendimiento) que podrían ser subcontratadas a un terceros. Esa parte se deja fuera del ámbito de los equipos, ya que es una cosa específica y separada. Por lo tanto, podría ser mejor abordar esa parte de una manera tradicional.
El proceso de mover las personas necesarias hacia una metodología ágil está sin duda asociado a varios desafíos. En primer lugar, se debe cambiar la cultura. Si los desarrolladores han estado trabajando en sus propias oficinas sin hablar con nadie más y de repente se los traslada a un ambiente compartido, se tiene que tener habilidades interpersonales para conseguir que esas personas puedan comunicarse.
Con el fin de superar esos desafíos, los gerentes de la empresa trataron de dar buenos ejemplos y convencer a la gente sobre la nueva forma de hacer las cosas. En primer lugar, se organizó una presentación para todo el personal y luego abordando el tema en grupos más pequeños, tratando de convencerlos y haciéndolos participar. Esto se hace muy bien por los “entrenadores agiles”, que tienen experiencia en ese tipo de cambio en organizaciones.

SP: ¿Tiene alguna sugerencia sobre cómo mejorar la productividad (en la industria del software) basándose en la experiencia personal de trabajo?

GH: La principal recomendación que daría es tratar de pensar de una manera más abierta. A menudo se sigue pensando demasiado acerca de la metodología que se utiliza y se olvida que todo lo que se necesita es tener personas capacitadas en el equipo. Por lo tanto, lo mejor que se puede hacer es dar la importancia correcta a la educación y al aumento del nivel de conocimiento.

12.3. CAPTURAS DEL SISTEMA

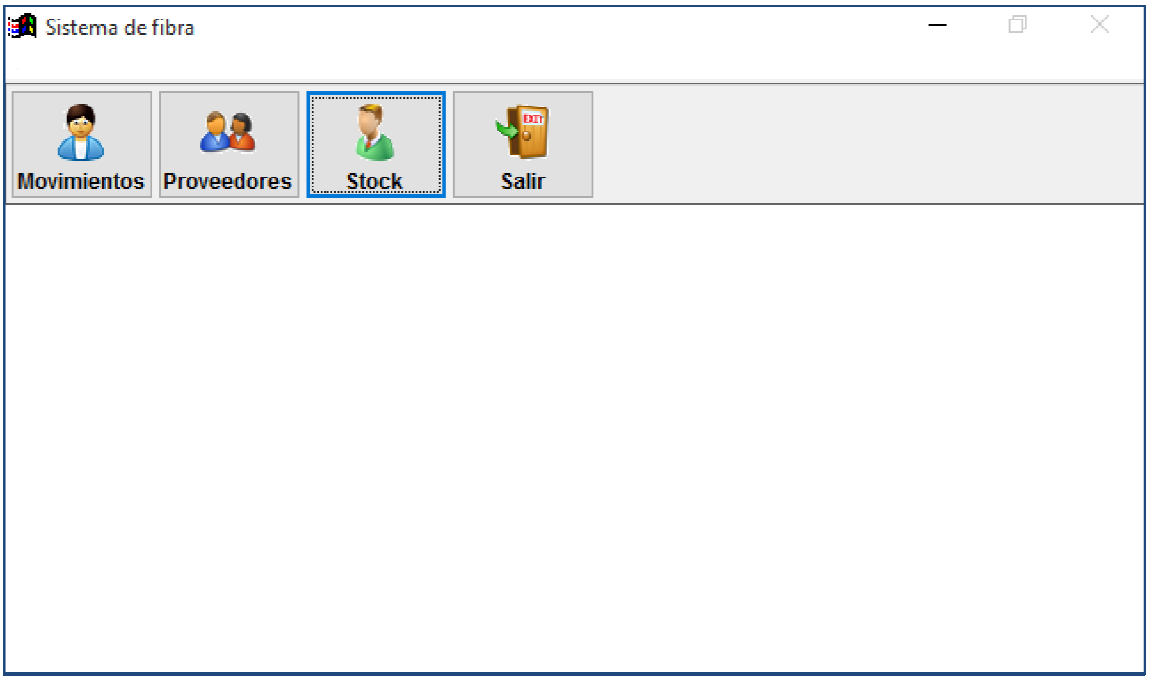


Figura 12.20 Sistema de Fibra – Pantalla inicial.

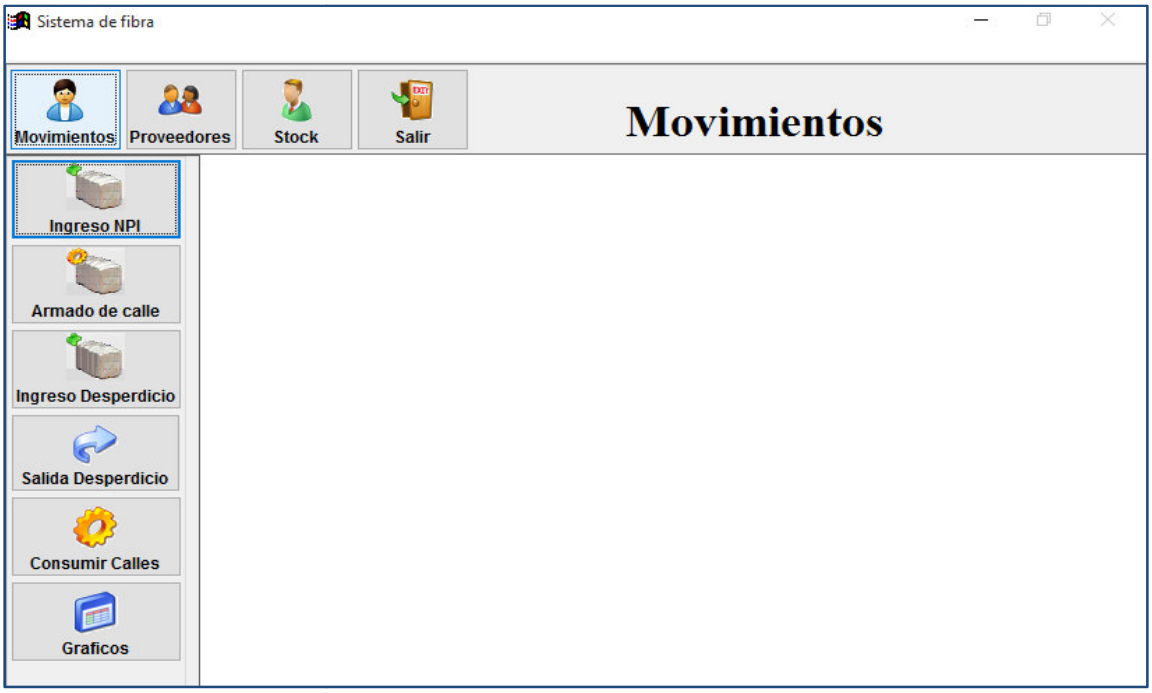


Figura 12.21 Sistema de Fibra – Modulo de Movimientos.

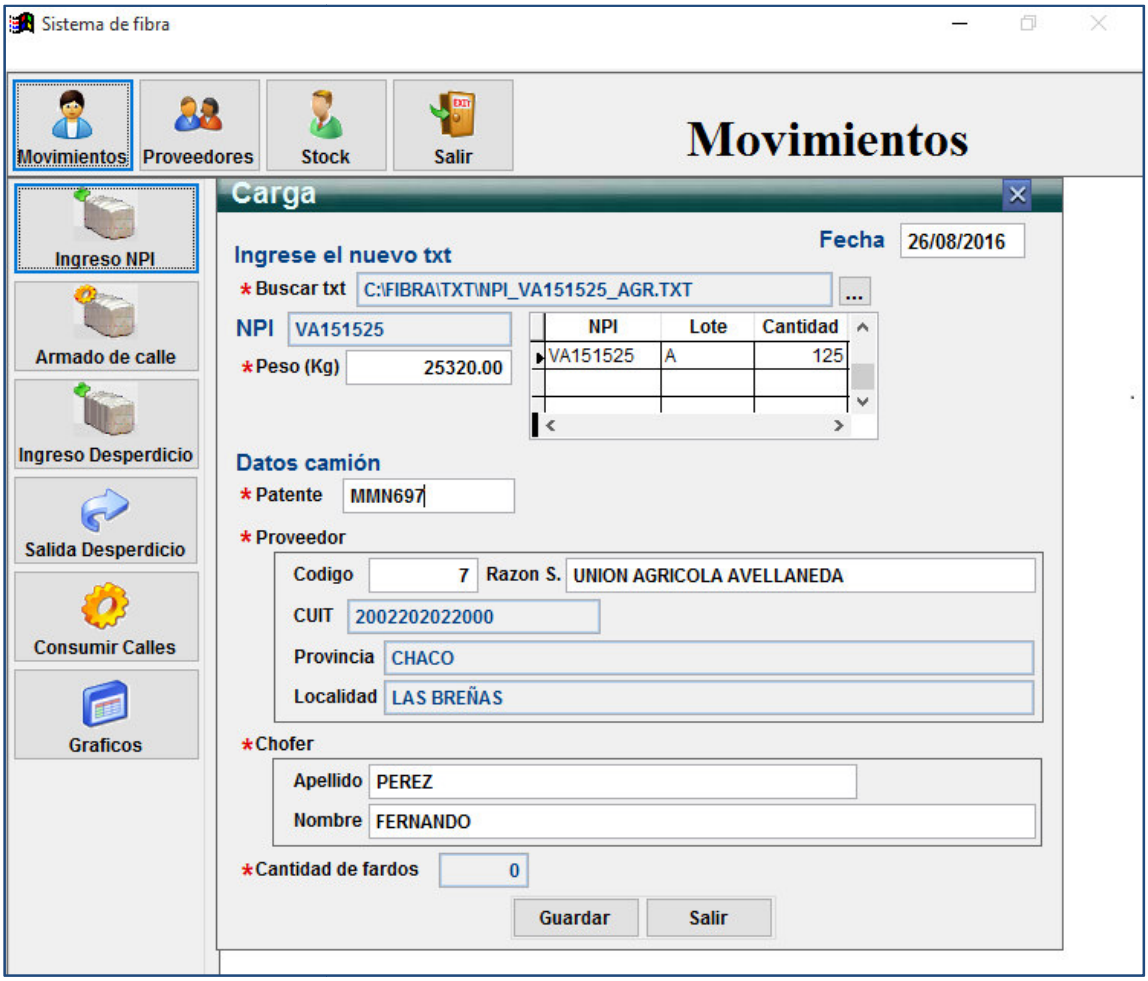


Figura 12.22 Sistema de Fibra – Ingreso de NPI.

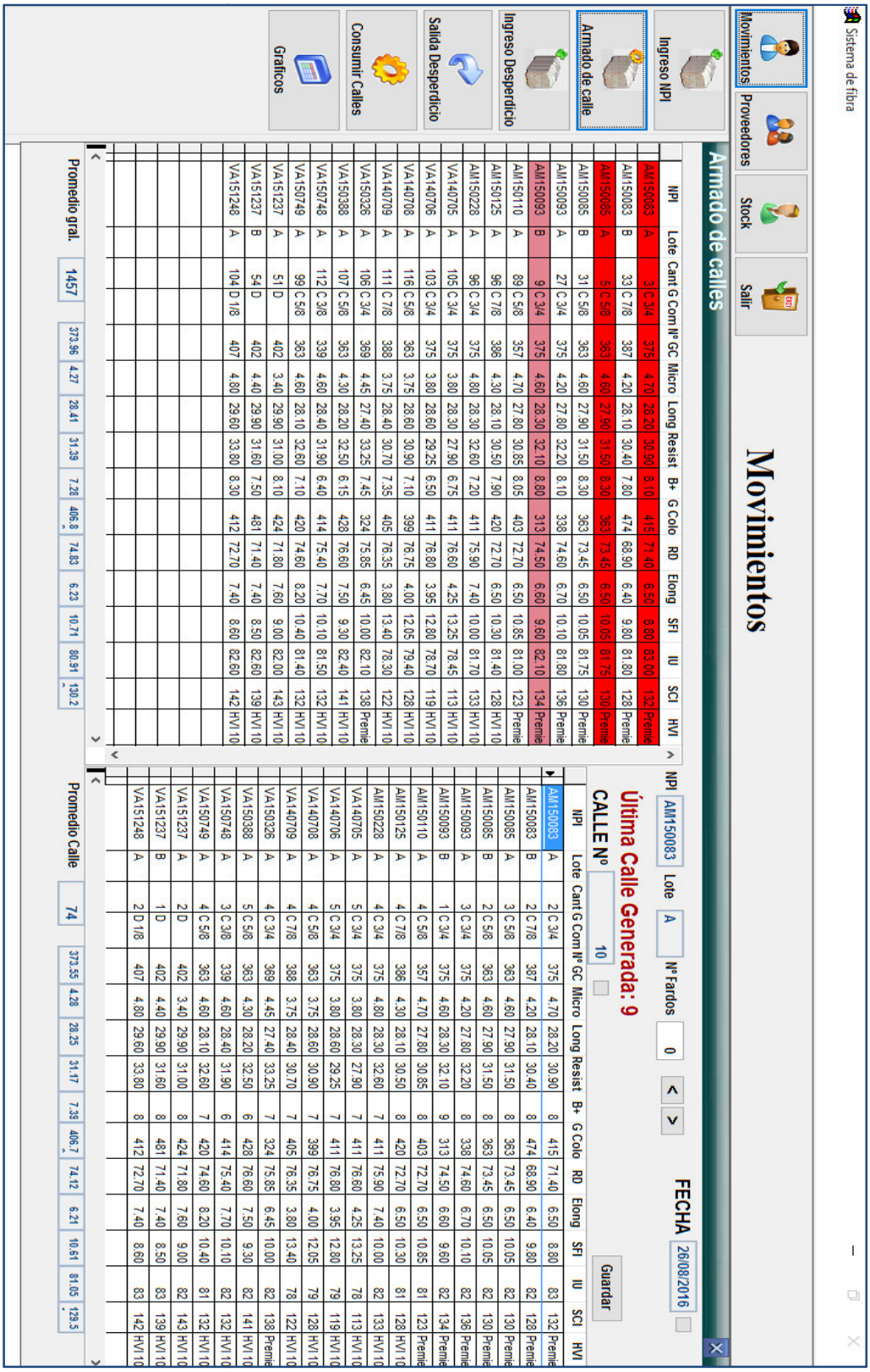


Figura 12.23 Sistema de Fibra – Modulo de armado de calles.

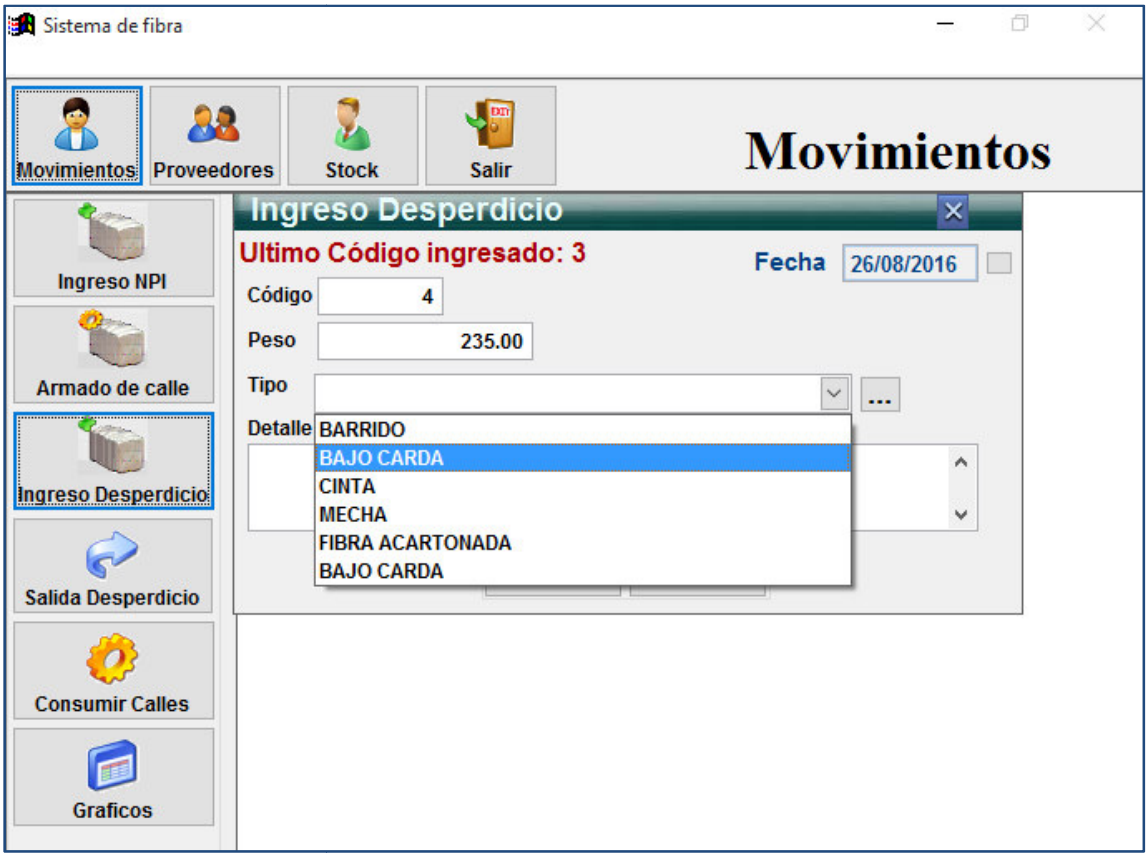


Figura 12.24 Sistema de Fibra – Ingreso de desperdicios.

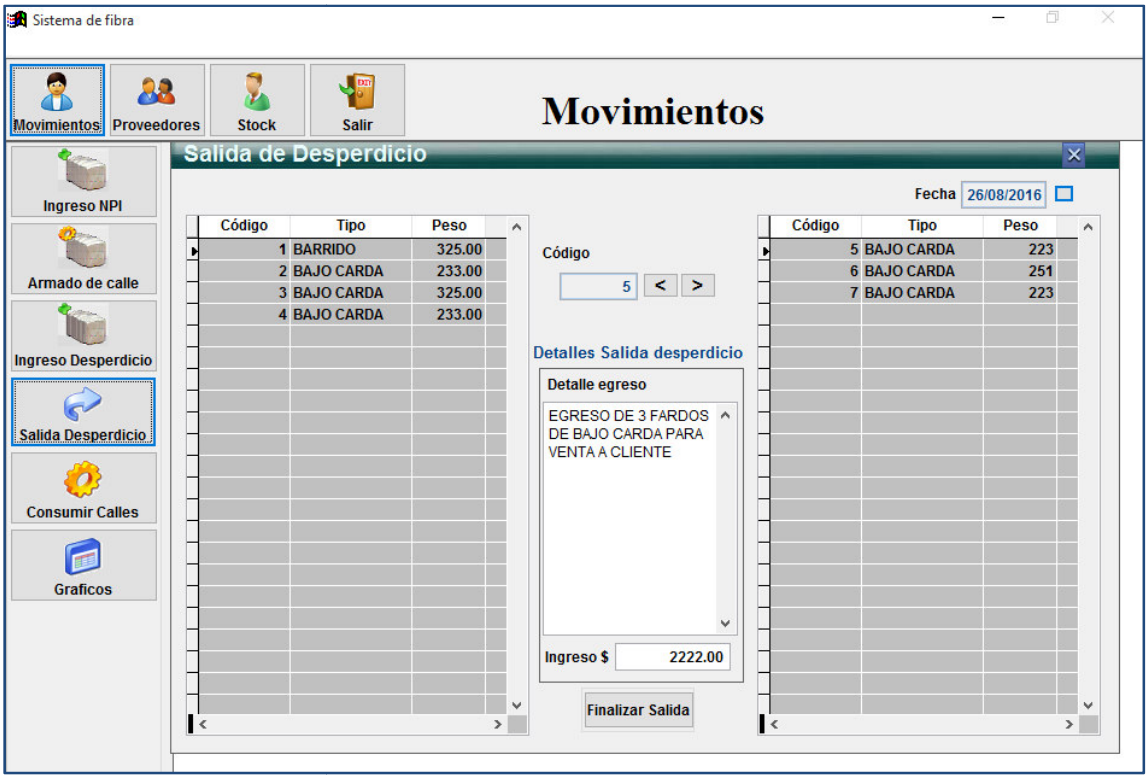


Figura 12.25 Sistema de Fibra – Salida de desperdicios.

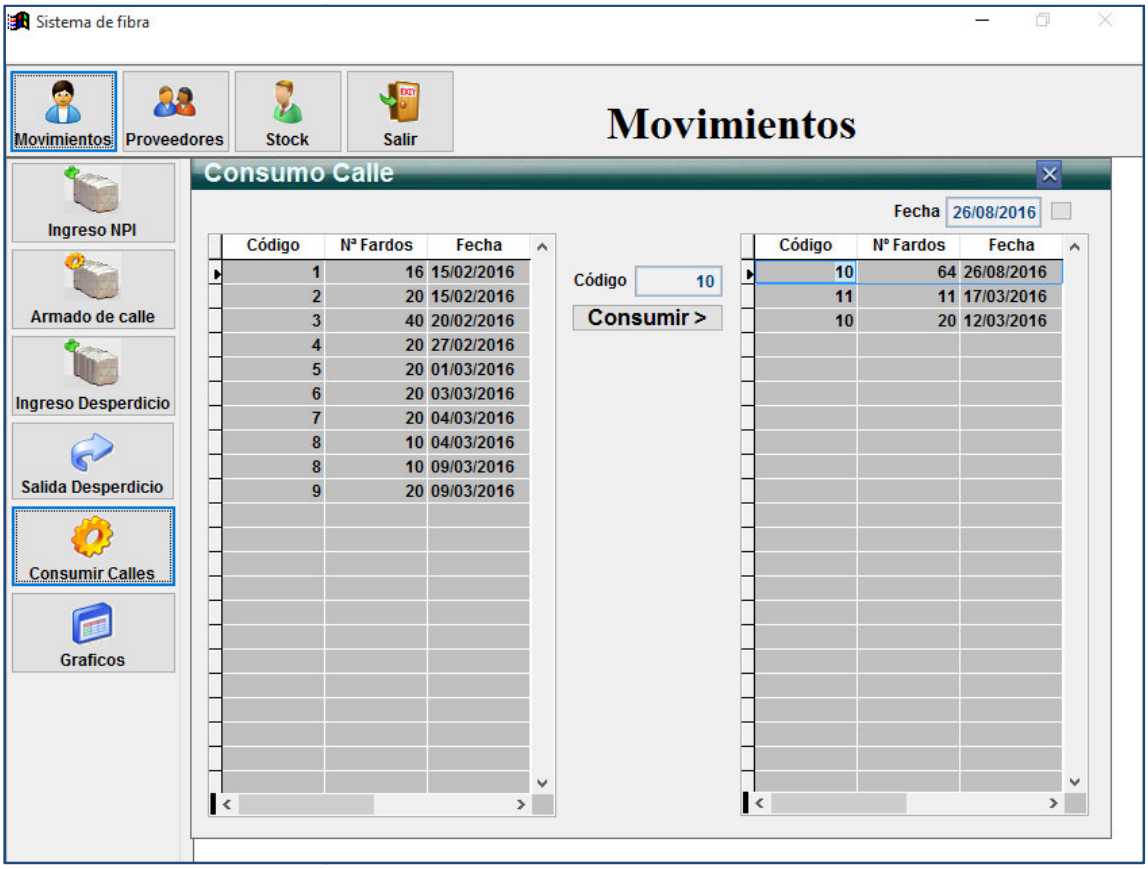


Figura 12.26 Sistema de Fibra – Baja de calles.

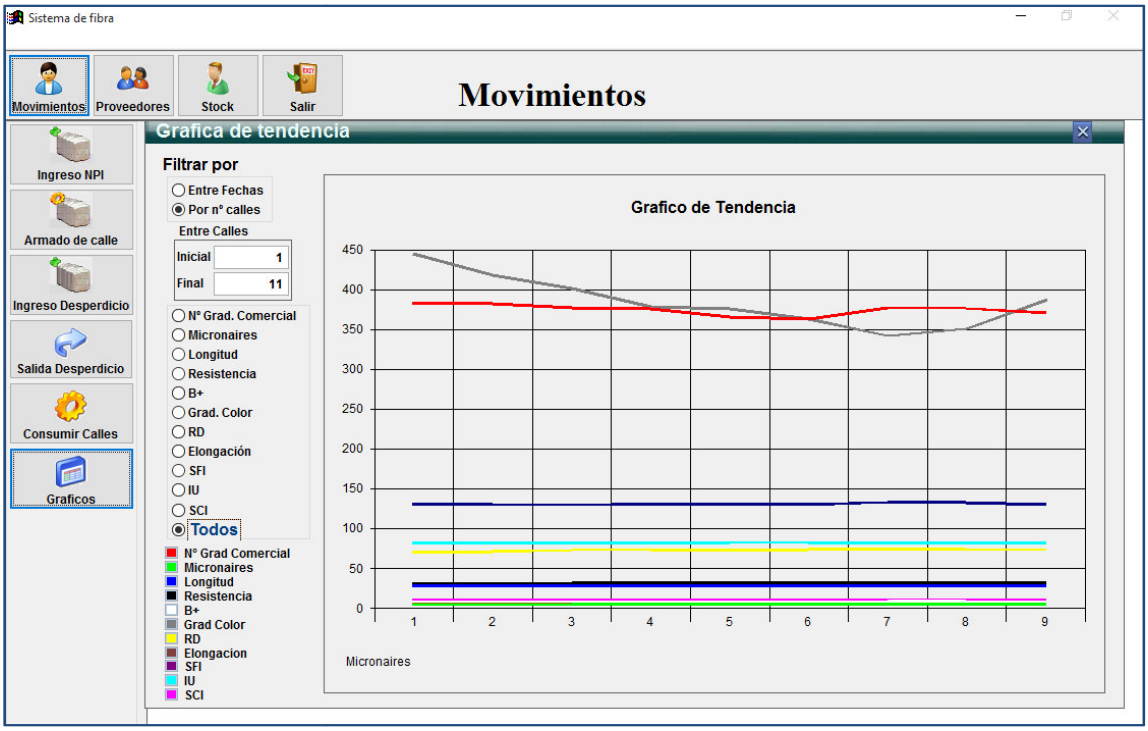


Figura 12.27 Sistema de Fibra – Gráficos de tendencia.

Figura 12.28 Sistema de Fibra – Módulo de proveedores

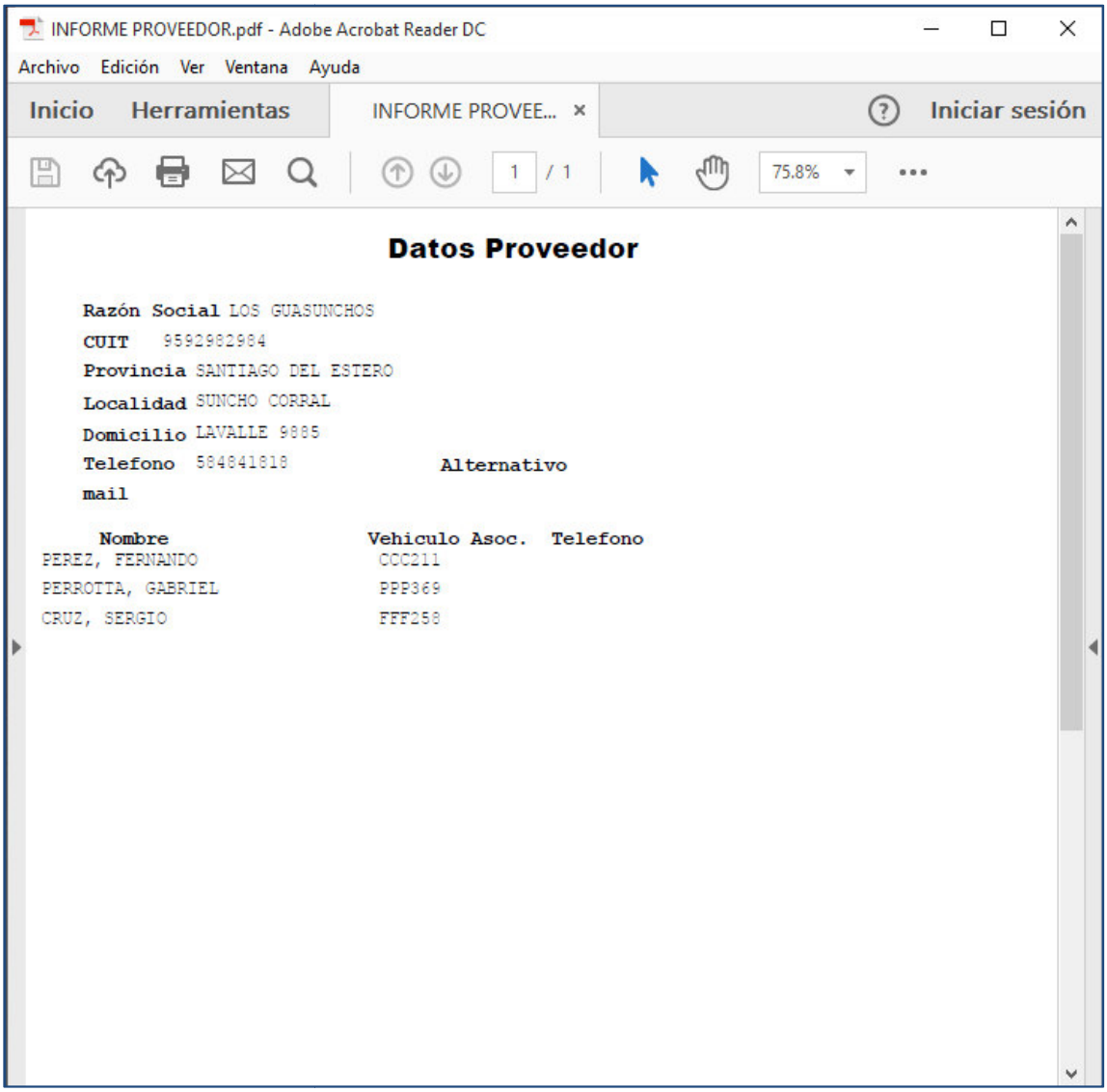


Figura 12.29 Sistema de Fibra – Reporte impreso de proveedor.

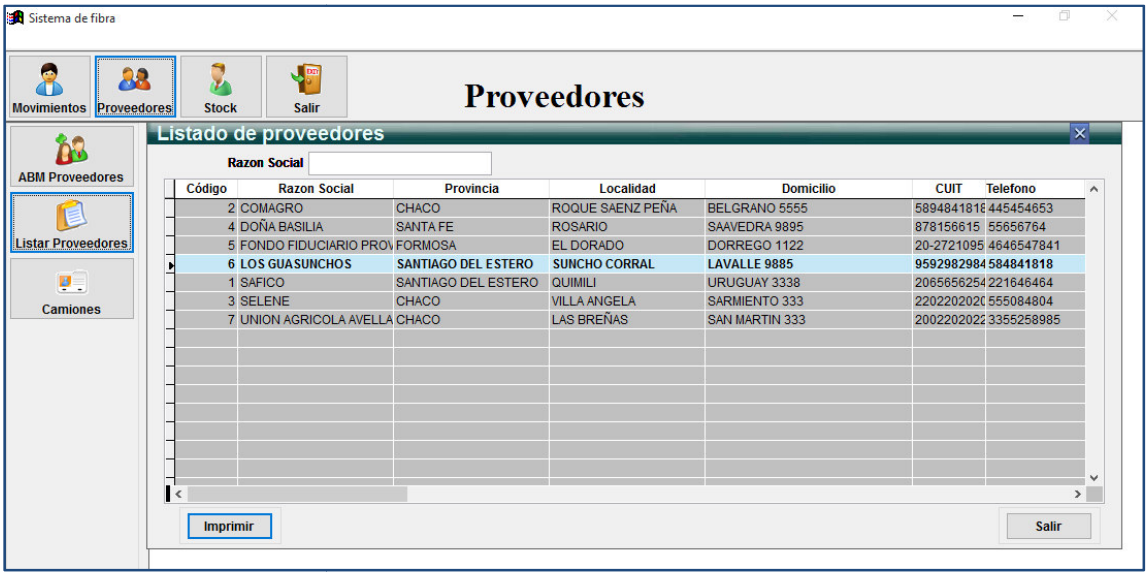


Figura 12.30 Sistema de Fibra – Listado de proveedores.

Figura 12.31 Sistema de Fibra – Consulta de camioneros.

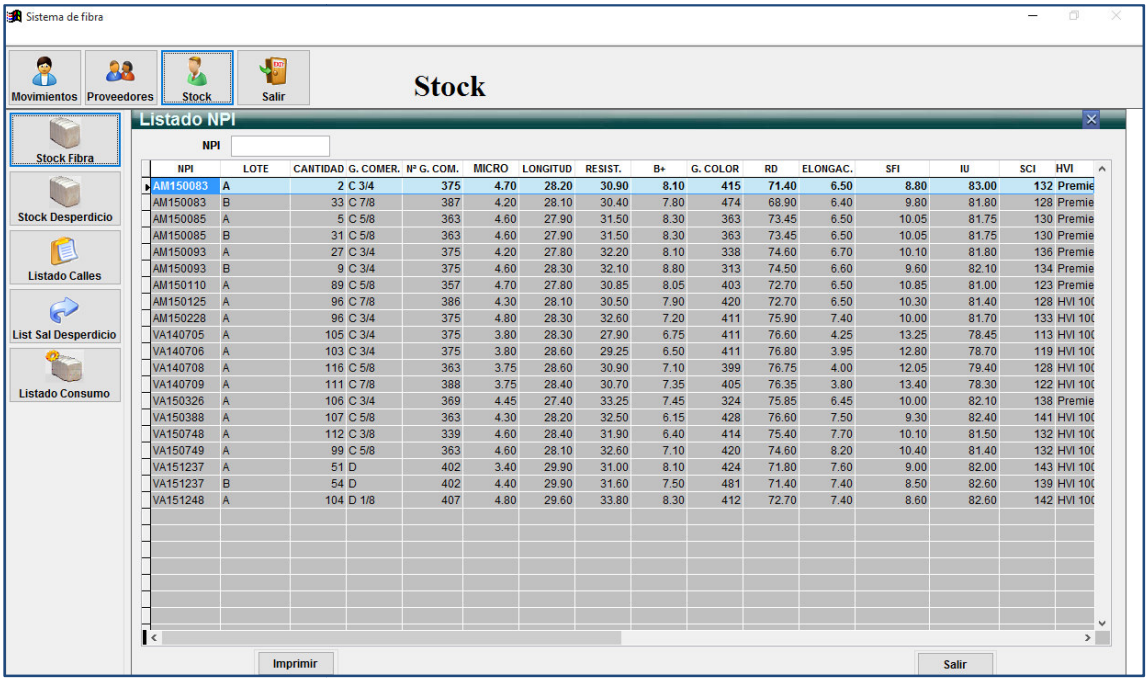


Figura 12.32 Sistema de Fibra – Pantalla de stock en planta.

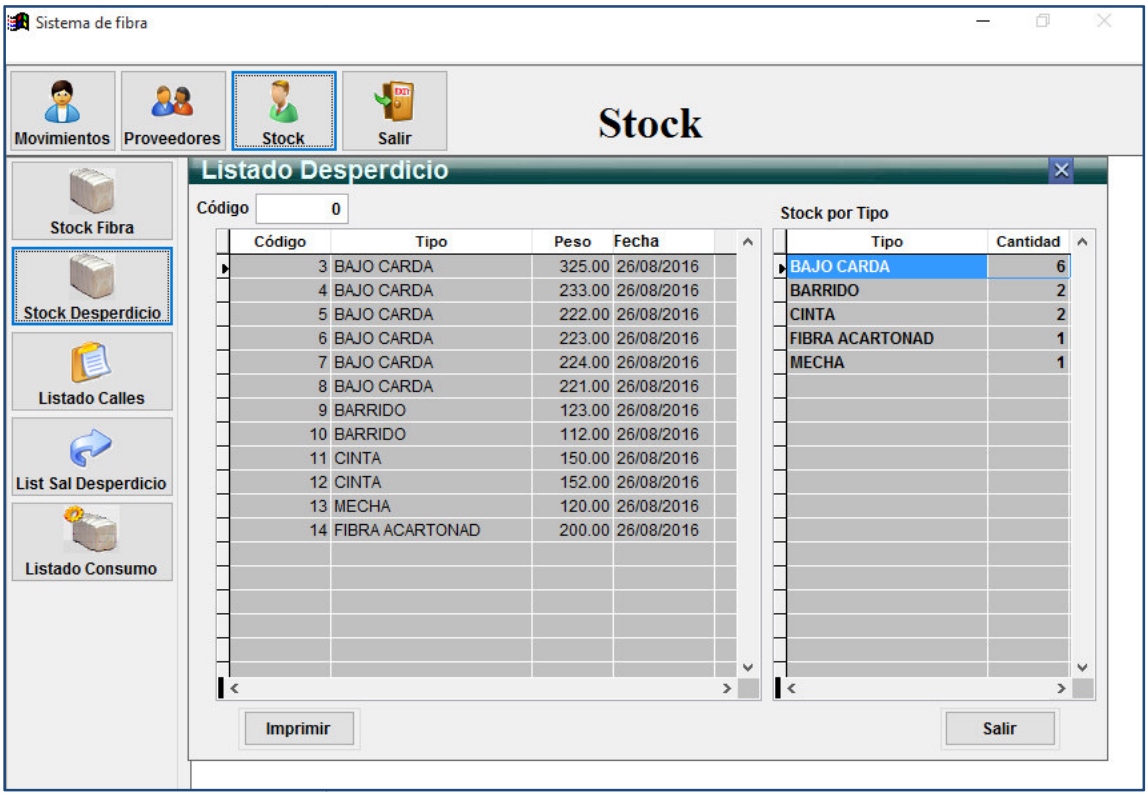
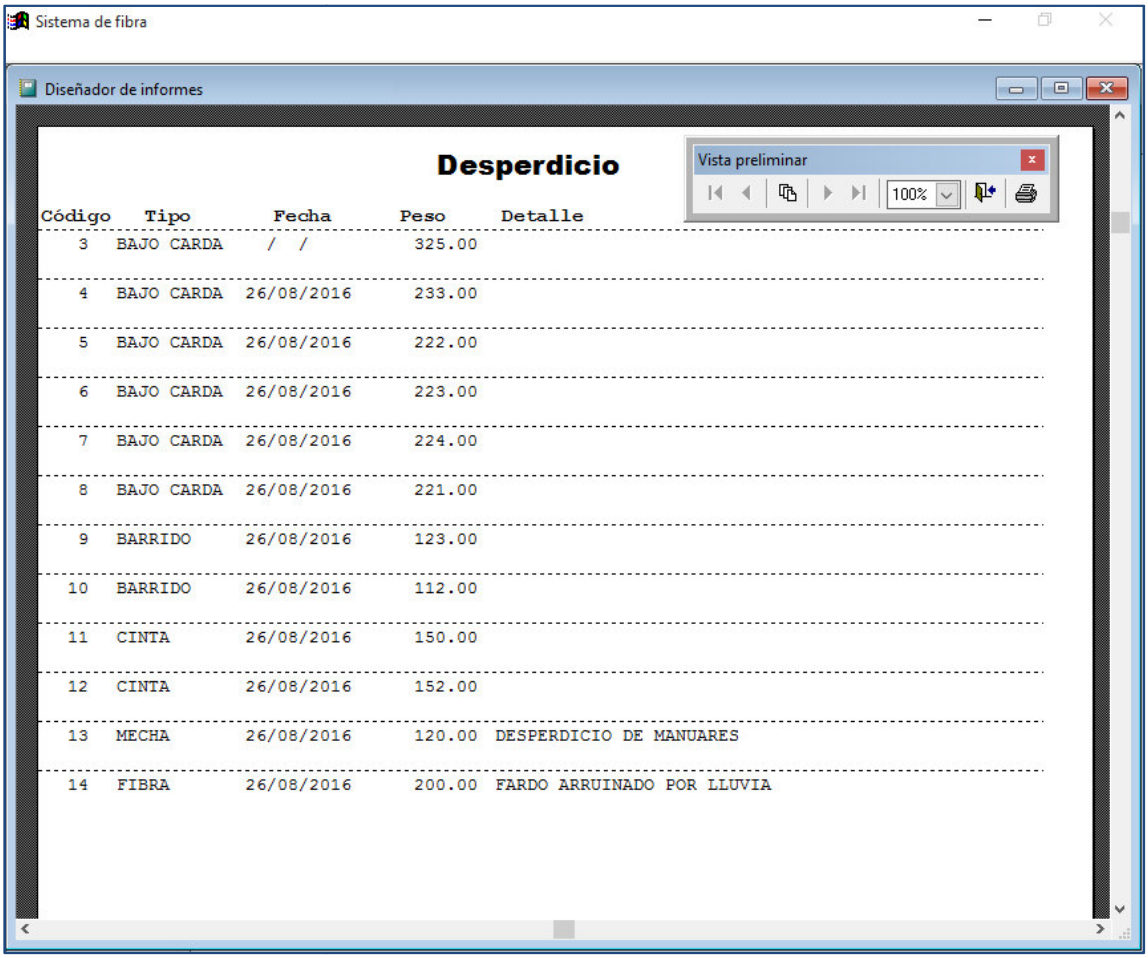


Figura 12.33 Sistema de Fibra – Pantalla de stock de desperdicio.



Desperdicio

Código	Tipo	Fecha	Peso	Detalle
3	BAJO CARDA	/ /	325.00	
4	BAJO CARDA	26/08/2016	233.00	
5	BAJO CARDA	26/08/2016	222.00	
6	BAJO CARDA	26/08/2016	223.00	
7	BAJO CARDA	26/08/2016	224.00	
8	BAJO CARDA	26/08/2016	221.00	
9	BARRIDO	26/08/2016	123.00	
10	BARRIDO	26/08/2016	112.00	
11	CINTA	26/08/2016	150.00	
12	CINTA	26/08/2016	152.00	
13	MECHA	26/08/2016	120.00	DESPERDICIO DE MANUARES
14	FIBRA	26/08/2016	200.00	FARDO ARRUINADO POR LLUVIA

Figura 12.34 Sistema de Fibra – Informe de stock de desperdicio.

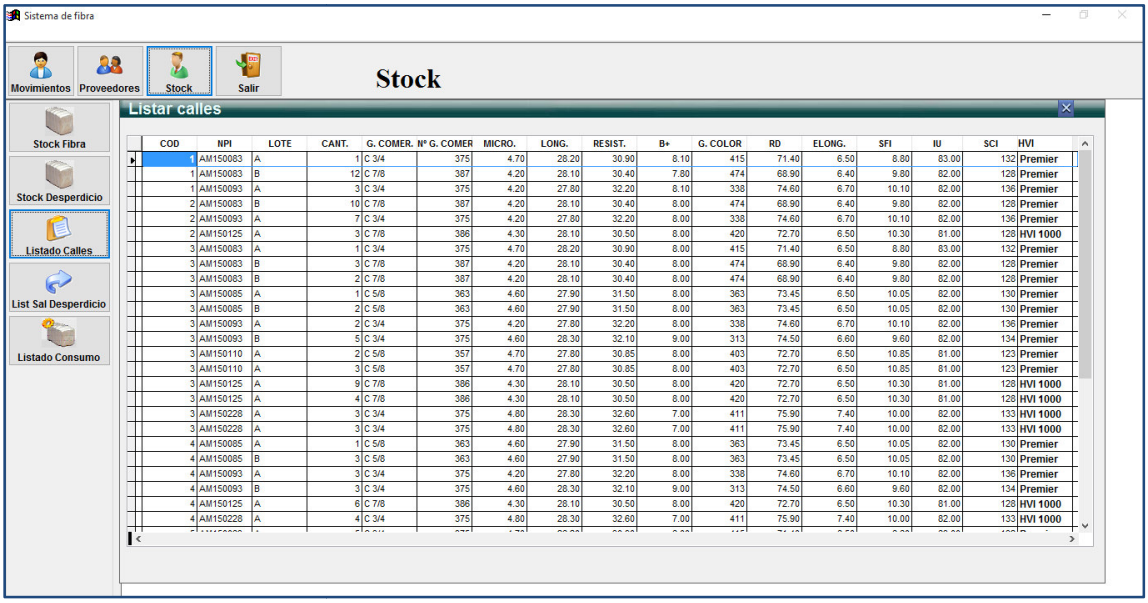


Figura 12.35 Sistema de Fibra – Listado de calles.

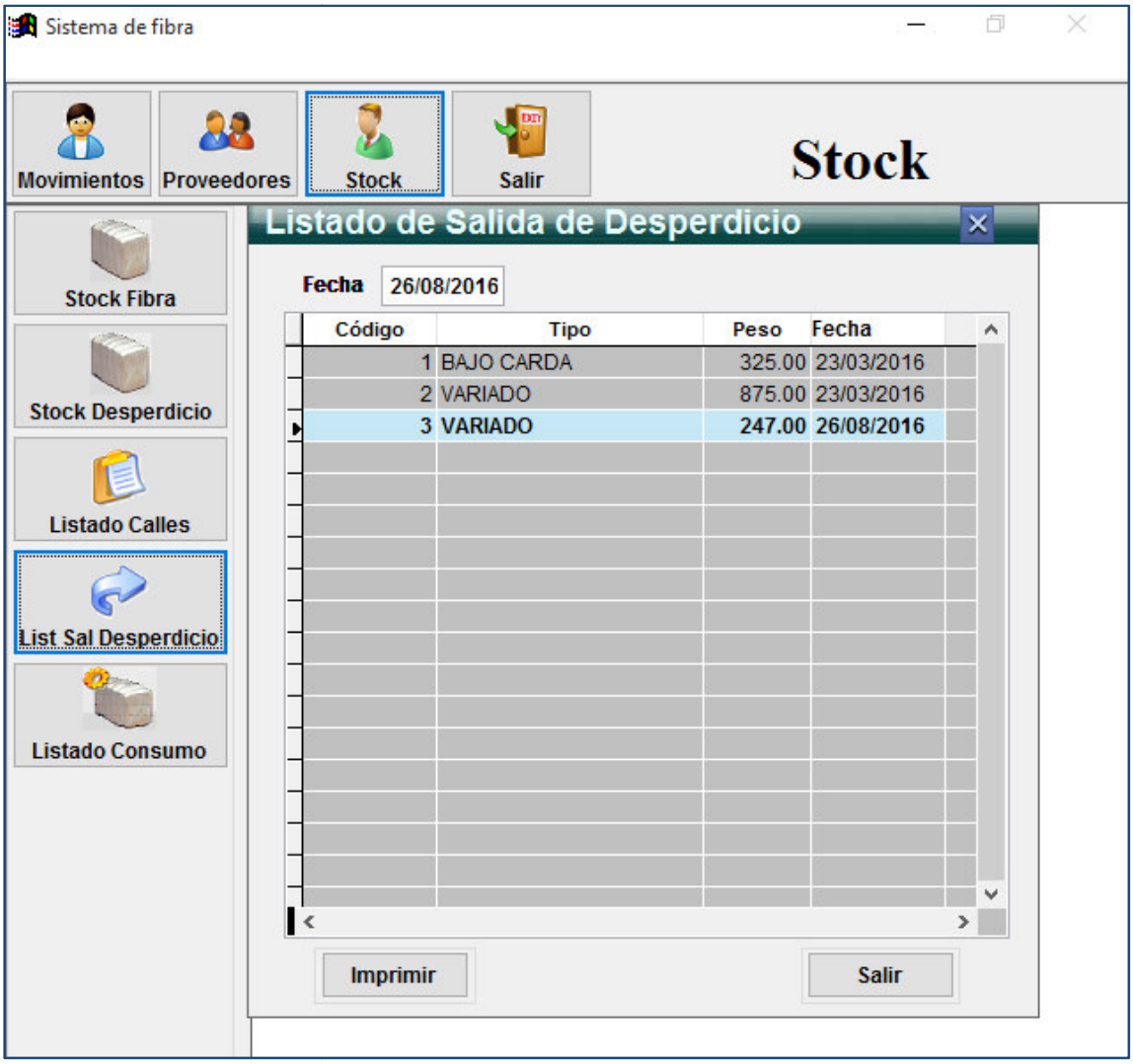


Figura 12.36 Sistema de Fibra – Listado de egresos de desperdicios.

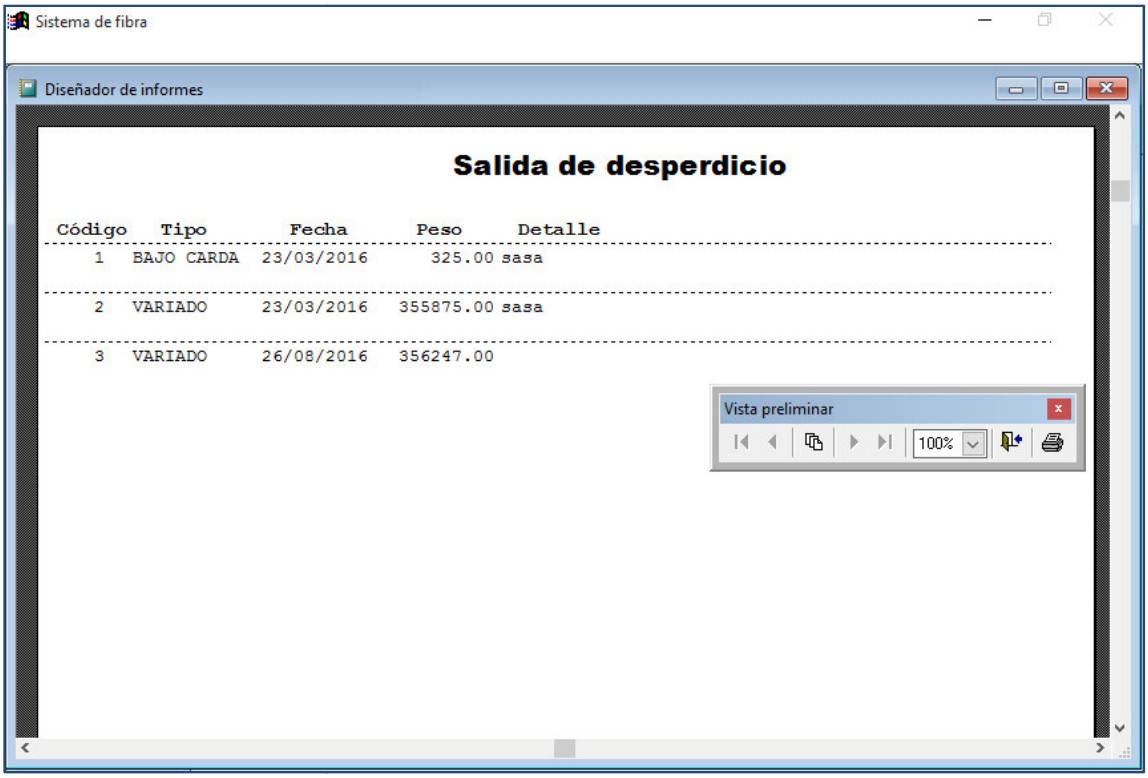


Figura 12.37 Sistema de Fibra – Reporte de salida de desperdicios.

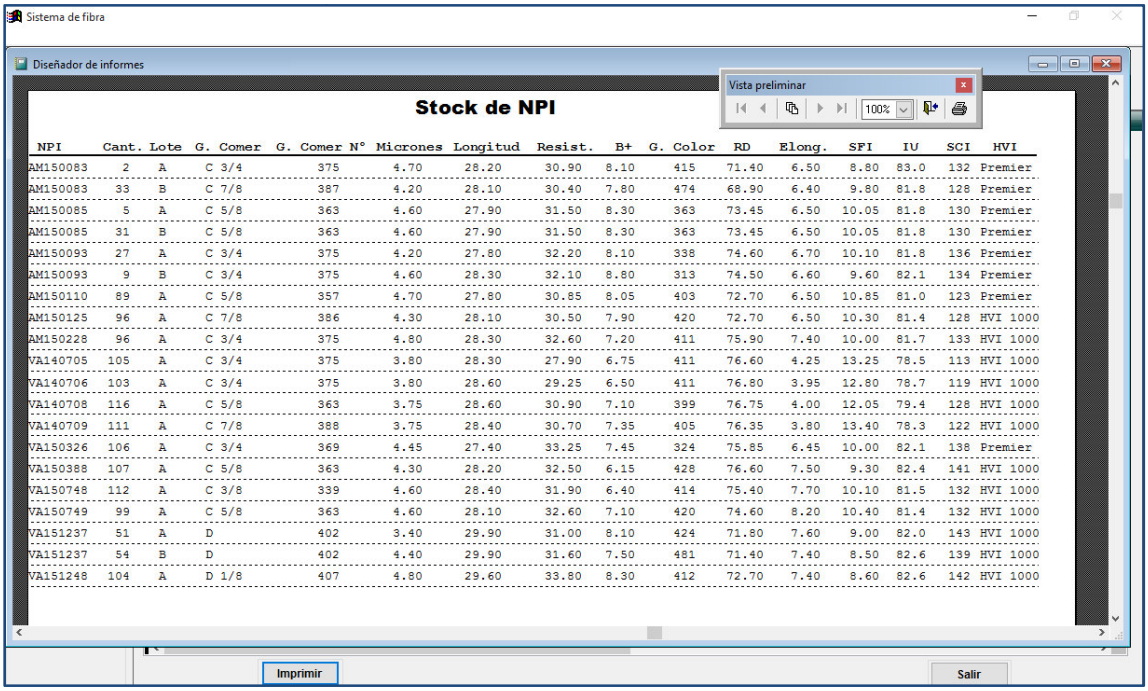


Figura 12.38 Sistema de Fibra – Reporte de stock de fibra en planta.

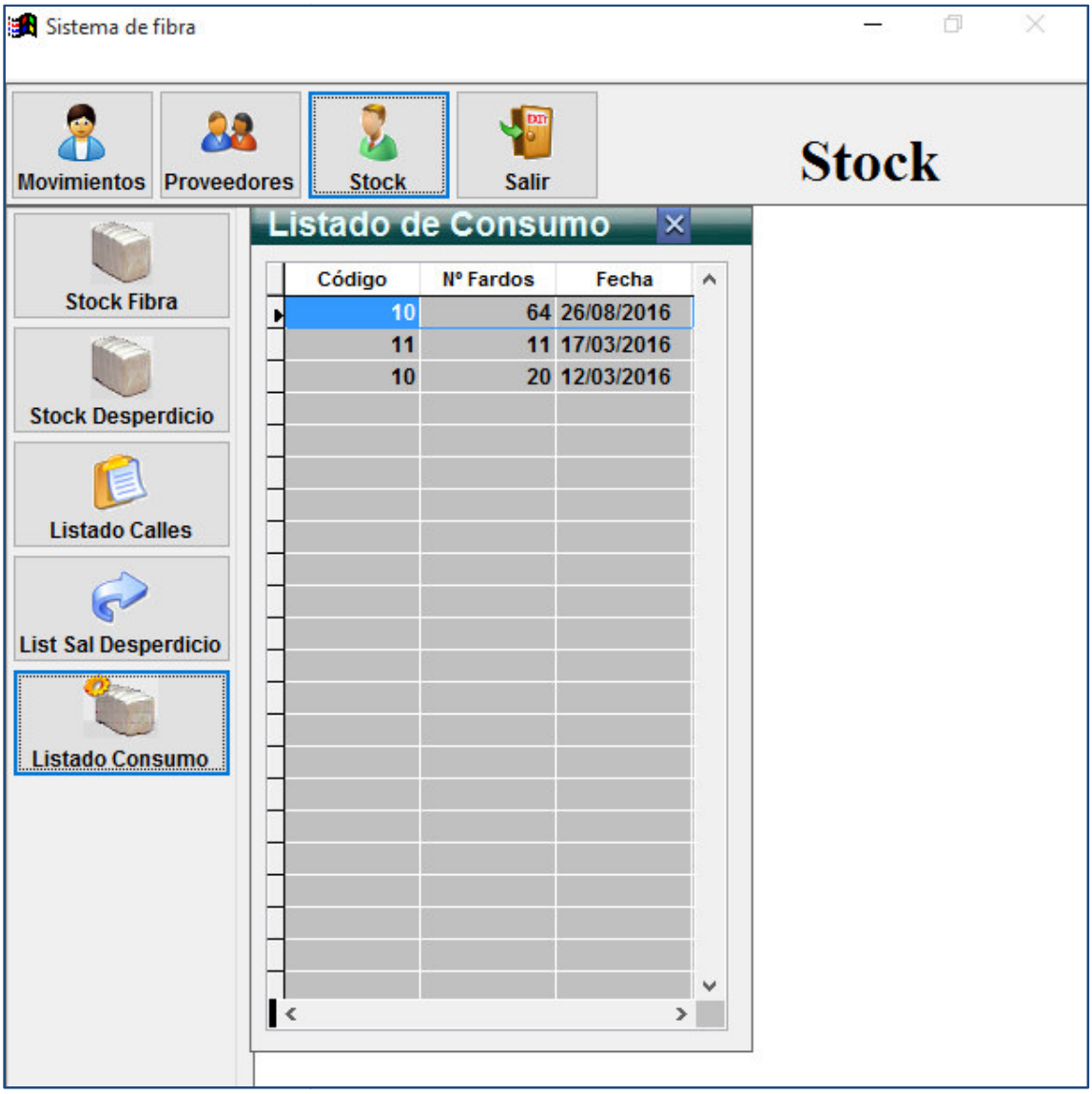


Figura 12.39 Sistema de Fibra – Listado de consumos de calles.

Figura 12.40 Sistema de Fibra – Salida de sistema.