Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

EVALUACIÓN DE LA METODOLOGÍA DE DESARROLLO DE *SOFTWARE* PROGRAMACIÓN EXTREMA

ALFREDO EDUARDO VALDÉS MATTA

ASESORADO POR INGA. ELIZABETH DOMÍNGUEZ ALVARADO

GUATEMALA, JUNIO DE 2004

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

EVALUACIÓN DE LA METODOLOGÍA DE DESARROLLO DE *SOFTWARE* PROGRAMACIÓN EXTREMA

TRABAJO DE GRADUACIÓN

PRESENTADO A JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

ALFREDO EDUARDO VALDÉS MATTA
ASESORADO POR: INGA. ELIZABETH DOMÍNGUEZ ALVARADO

AL CONFERÍRSELE EL TÍTULO DE
INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, JUNIO DE 2004

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Sydney Alexander Samuels Milson
VOCAL I	Ing. Murphy Olympo Paiz Recinos
VOCAL II	Lic. Amahán Sánchez Álvarez
VOCAL III	Ing. Julio David Galicia Celada
VOCAL IV	Br. Kenneth Issur Estrada Ruiz
VOCAL V	Br. Elisa Yazminda Vides Leiva
SECRETARIO	Ing. Pedro Antonio Aguilar Polanco

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Sydney Alexander Samuels Milson
EXAMINADOR	Ing. César Fernández Cáceres
EXAMINADOR	Ing. Ricardo Morales Prado
EXAMINADOR	Ing. Guillermo Rafael Sánchez Barrios
SECRETARIO	Ing. Pedro Antonio Aguilar Polanco

HONORABLE TRIBUNAL EXAMINADOR

Cumpliendo con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

EVALUACIÓN DE LA METODOLOGÍA DE DESARROLLO DE *SOFTWARE* PROGRAMACIÓN EXTREMA

tema que me fuera asignado por la Escuela de Ingeniería en Ciencias y Sistemas de la Facultad de Ingeniería con fecha febrero de 2003.

Alfredo Eduardo Valdés Matta

Guatemala, 22 de febrero de 2004

Ing. Carlos Alfredo Azurdia Morales
Coordinador de Privados y Trabajos de Graduación
Escuela de Ingeniería en Ciencias y Sistemas
Facultad de Ingeniería
Universidad de San Carlos de Guatemala

Ing. Azurdia:

Por medio de la presente hago de su conocimiento que he tenido a bien revisar el trabajo de graduación de ALFREDO EDUARDO VALDÉS MATTA, titulado "EVALUACIÓN DE LA METODOLOGÍA DE DESARROLLO DE *SOFTWARE* PROGRAMACIÓN EXTREMA", por lo cual me permito recomendar dicho trabajo final para la respectiva revisión por parte de la comisión de trabajos de graduación de la escuela de Ciencias y Sistemas.

Sin otro particular, me suscribo atentamente,

Ing. Elizabeth Domínguez Alvarado
ASESOR

Agradezco a todos los que me han ayudado
para lograr mis metas.

ÍNDICE

ÍNDICE DE ILUSTRACIONES

FIGURAS

GLOSARIO

Actor	Es una idealización de una persona externa, de un proceso, o de una cosa que interactúa con un sistema, un subsistema, o una clase.
Administración	Cualidad de planificar, organizar, dirigir y controlar una actividad
Capacidad	Habilidad que tienen un sistema para alcanzar las sus metas y objetivos.
Caso de prueba	Un conjunto automatizado de estímulos y respuestas para el sistema. Cada caso de prueba dejará el sistema en el estado en que se lo encontró, así los casos de prueba funcionan independientemente unos de otros.
Caso de uso	Secuencia de interacciones entre un sistema y alguien o algo que usa alguno de sus servicios. Fragmento de funcionalidad que proporciona al usuario un resultado importante

Clase	Conjunto de objetos que comparten una estructura común y un comportamiento común.
D.E.R.C.A.S.	Documento de especificación de requerimientos y criterios de aceptación del sistema.
Episodio de programación	Es donde el programador implemente una tarea de ingeniería (la menor unidad de planificación) y lo integración el resto del sistema
Estado	Un estado describe un período de tiempo durante la vida de un objeto de una clase
Evento	Es una ocurrencia significativa que tiene una localización en el tiempo y espacio.
Factor de carga	La relación entre el tiempo de programación ideal y el tiempo transcurrido.
Historia	Algo que el cliente quiere que el sistema haga. La historia se debería estimar entre una y cinco semanas de programación ideal.

Iteración	Un período de tiempo de una a cuatro semanas en el cual se realiza una actividad del desarrollo del *software*.
Liderazgo	Consiste en influir a los demás para que se esfuercen en lograr una o más metas.
Metáfora	Una historia que todos (clientes, programadores y directores de proyecto) pueden contar sobre cómo funciona el sistema.
Método	Operación sobre un objeto, definida como parte de la declaración de una clase.
Modificabilidad	Factor de calidad en el cual se aprecia la facilidad de un producto para adaptarse al cambio de especificaciones.
Modularidad	Independencia funcional de los componentes del programa.
Objeto	Algo a lo cual se le pueden hacer cosas, tiene estado, comportamiento e identidad.

Poder	Capacidad de una persona para influir en el comportamiento de los demás.
Poder de retribución	Es la influencia que se deriva de la capacidad del líder para satisfacer las necesidades de los seguidores.
Poder experto	Es la influencia basada en los conocimientos y las competencias del líder.
Programación en parejas	Técnica de programación en la que dos personas programan en una misma computadora para minimizar riesgos.
Prototipo	Modelo del sistema final que no contiene todas las características del sistema final.
Recodificación	Un cambio en el sistema que lo deja sin cambiar su comportamiento, pero mejora alguna característica no funcional como la sencillez, flexibilidad, comprensión y rendimiento).
Reglas del negocio	Determinan políticas y estructuras de la información para cada proceso del negocio.

Requerimiento	Característica que debe incluirse en un nuevo sistema.
Reusabilidad	Factor de calidad en la cual se mide la facilidad del *software* para ser reutilizado en todo o en parte para nuevas aplicaciones.
Rol	Papel que un actor juega dentro de un sistema.
Sistema	Conjunto de componentes que interaccionan entre si para lograr un objetivo común.
Stakeholder	Todos aquellos actores que intervienen en el proceso de creación de un sistema.
Teoría Y	Conjunto de propuestas y opiniones en que se considera un planeamiento de la administración en términos de liderazgo y delegación de autoridad basada en un punto de vista positivo sobre la naturaleza humana.
Tiempo de estimación ideal	El número ideal de semanas de programación que el equipo puede producir en una cantidad de tiempo determinada.

Tiempo ideal de ingeniería	Estimación de tiempo para la programación de una historia si no se tiene interrupciones o reuniones.
XP	Programación extrema.

RESUMEN

Una metodología de desarrollo de *software,* asegura al desarrollador de un proyecto de *software* a identificar los procesos necesarios para crear una solución y, de esta forma poder organizar al equipo de trabajo para tener un producto de calidad.

El ciclo de vida clásico real tiene los procesos de análisis, diseño, codificación, pruebas y mantenimiento, los cuales se deben de seguir de forma secuencial. Esta metodología ya no se aplica a proyectos de *software* pero si a pequeños módulos del sistema.

Otra metodología de desarrollo de *software* es también, llamada Proceso Unificado, el cual se basa en prácticas como: la administración de requerimientos, la verificación de la calidad, el desarrollo iterativo, el modelado visual, el control de cambios y la utilización de una arquitectura con componentes. Esta metodología se divide en cuatro fases: (inicio, elaboración, construcción, transición). A lo largo del proyecto, se trabaja en diferentes disciplinas, el esfuerzo que se realiza en cada una de ellas, depende de la fase en la que se encuentra y el avance de que se lleva un una iteración.

Una metodología ágil que se ha utilizado para el desarrollo de *software* es la programación extrema que fue diseñada por Kent Beck para responder a los cambios de los requerimientos y ser utilizada con equipos de desarrollo pequeños y medianos. Las prácticas que se utilizan con esta metodología son: la utilización de versiones pequeña, la elaboración de un diseño sencillo, el manejo pruebas continuas, la utilización de la recodificación, la programación en parejas, la aceptación de la propiedad colectiva del código fuente, el uso de la integración continua, el trabajo de cuarenta horas semanales y la utilización de estándares.

No todos los equipos de desarrollo pueden utilizar la programación extrema, y tampoco se puede utilizar la programación extrema en todos los proyectos. Existen varios factores que hay que tomar en cuenta para poder migrar a la programación extrema. Se puede mencionar el tamaño del equipo de desarrollo, el tipo de proyecto que se va a trabajar, la forma de liderazgo del equipo, las instalaciones del área de desarrollo, la actitud del cliente entre otros.

OBJETIVOS

General

Evaluar la metodología de desarrollo de *software* Programación extrema.

Específicos

1. Aclarar qué es una metodología de desarrollo de *software*.

2. Especificar qué es el ciclo de vida del *software*.

3. Identificar conceptos de Programación extrema.

4. Conocer técnicas para implantar la metodología de Programación extrema en un equipo de desarrollo de *software*.

INTRODUCCIÓN

El desarrollo de *software* ha utilizado una serie de metodologías para crear un producto que satisfaga las necesidades del cliente. Estas metodologías han estado cambiando, tratando de evitar que los proyectos se atrasen, los costos de producción aumenten, existan errores de programación y la insatisfacción del cliente.

Una metodología con poca historia es la Programación extrema (XP), la cual, es un proceso de desarrollo de *software* para equipos pequeños que se enfrentan a requerimientos cambiantes o difíciles de identificar. Esta metodología permite al cliente tener una participación más activa en el desarrollo de producto final, por lo cual, es más fácil identificar los requerimientos del sistema, y de esta forma realizar un cronograma más exacto.

Conociendo la forma en que trabaja la programación extrema, se podrá apreciar la factibilidad de introducir esta metodología a un grupo de desarrolladores nacionales, la forma en que se debe administrar e identificar que habilidades se deben crear y cuales aprovechar.

1 METODOLOGÍA DE DESARROLLO DE *SOFTWARE*

La ingeniería de la programación se define como la disciplina tecnológica relacionada con la producción sistemática y el mantenimiento de productos de *software* que son desarrollados y modificados en el tiempo previsto y dentro de los costes estimados, si el producto esta en su ciclo de vida, la ingeniería de la programación existe.

Una metodología de desarrollo de *software* asegura al desarrollador de un proyecto de *software* a identificar los procesos necesarios para crear una solución y de esta forma poder organizar al equipo de trabajo para poder tener un producto de calidad.

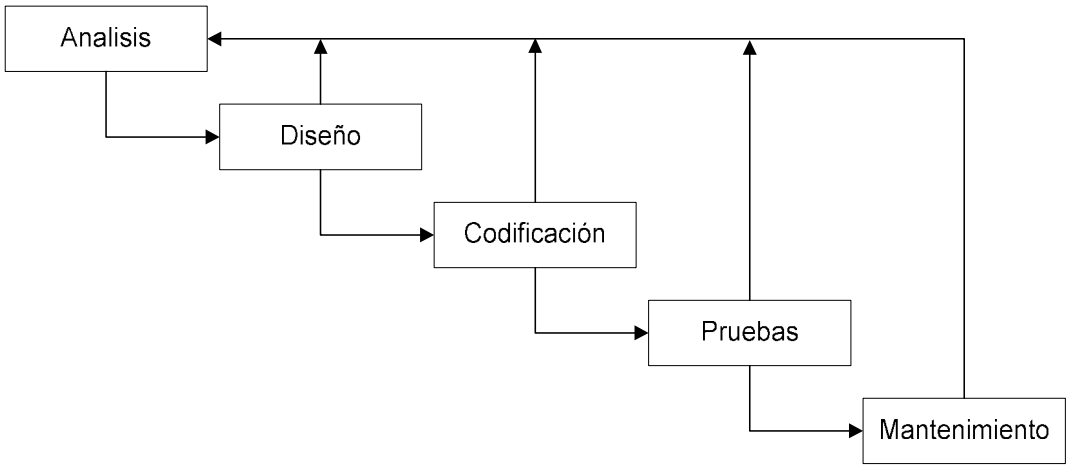
La metodología de desarrollo de *software* se selecciona dependiendo la complejidad de un proyecto y la experiencia que se tenga en el campo y la experiencia del equipo de trabajo con las diferentes metodologías existente.

Esta metodología, también es conocida como ciclo de vida del *software*, que es la sucesión de etapas por las que atraviesa un producto de *software* a lo largo de su desarrollo y existencia. Se puede destacar las siguientes etapas: el análisis de sistemas, diseño de sistemas y codificación entre otras.

1.1 Ciclo de vida clásico real

Una de las primeras secuencias de procesos para crear *software* utilizadas por los desarrolladores fue el ciclo de vida clásico real. El diagrama de flujo de dicha secuencia se puede ver en la siguiente gráfica.

Figura 1. Diagrama de flujo para el ciclo de vida clásico real



1.1.1 Análisis

El análisis de sistemas es el proceso de clasificación e interpretación de hechos, diagnósticos de problemas y empleo de la información para recomendar mejoras al sistema. Especifica qué es lo que hace el sistema.

La determinación de requerimientos estudia la forma en que trabaja un sistema, comprenderlo y poder identificar dónde se pueden efectuar mejoras. Todas aquellas características que se deben incluir en un sistema son conocidas como requerimientos.

En el análisis no se establece cómo se cumplirá los requerimientos o la forma en que se implantará la aplicación.

Debido a que el análisis es la piedra angular de todo el proyecto, es necesario documentar de una forma clara todos los hechos que fueron obtenidos del sistema, para esto se genera una serie de documentos que ayudan comprender las tareas del proyecto.

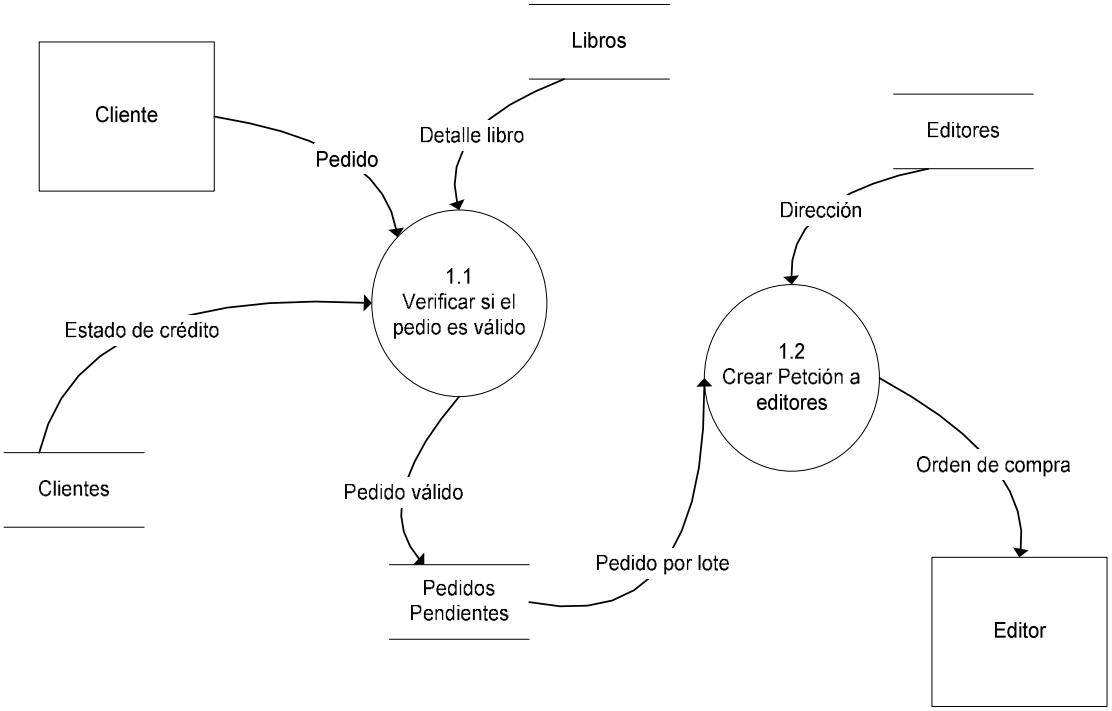
1.1.1.1 Diagrama de flujo de datos DFD

Método gráfico que sirve para modelar el flujo de información dentro del sistema. El sistema debe estar completamente descrito al tener el conjunto de todos los DFD creados.

Los DFD muestran las transformaciones aplicadas a los datos desde la entrada hasta la salida, también permite el particionamiento del sistema en diferentes niveles de detalle.

En la gráfica se aprecia un ejemplo DFD aplicado al flujo de pedidos de libros.

Figura 2. Ejemplo de un DFD



1.1.1.2 Diccionario de datos

En este documento se tiene una definición detallada de cada elemento del sistema. Se debe incluir a todos los flujos de datos y almacenes de datos que aparecen en el DFD.

La nomenclatura que se utiliza para elaborar un diccionario de datos se puede ver en la gráfica a continuación.

Figura 3. Nomenclatura de un diccionario de datos

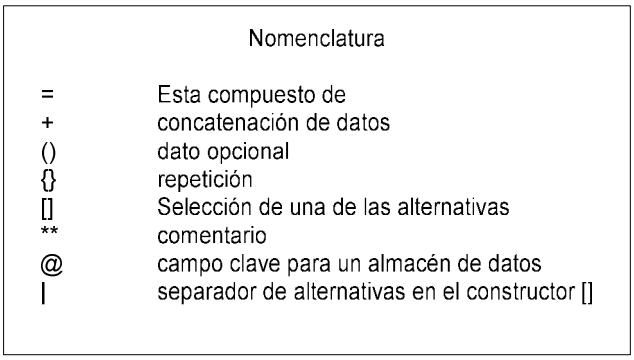
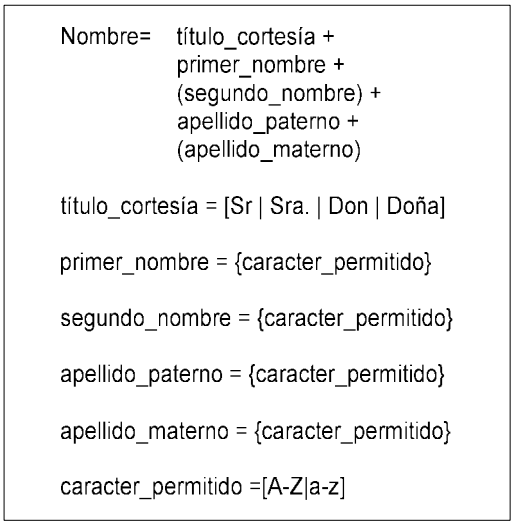


Figura 4. Ejemplo de un diccionario de datos



La anterior gráfica muestra un ejemplo de un diccionario de datos que se puede utilizar para identificar el nombre de una persona.

1.1.1.3 Especificación de procesos

Se debe explicar la forma de trabajo lógica interna de cada proceso del DFD, o sea, dar una definición de qué debe hacerse para transformar las entradas en salidas. Esto se realiza ya sea por medio de un lenguaje estructurado, una tabla de decisiones, un diagrama de flujo por mencionar una de las herramientas disponibles para este hecho.

A continuación se puede ver un DFD de una validación de crédito y su especificación correspondiente.

Figura 5. Ejemplo de un proceso a especificar

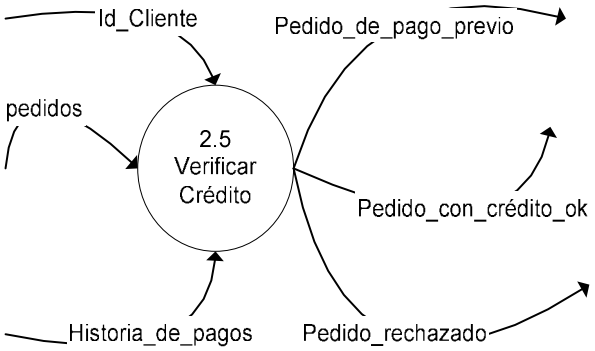


Figura 6. Ejemplo de la especificación de un proceso

Nombre Proceso:	Verificar_Crédito
Número:	3.5
Definición:	Decidir tratamiento de pago para pedidos. Sin previo pago o si debe pedirse el pago al cliente.
Entradas:	Id_Cliente pedidos historia_de_pagos
Salidas:	Pedidos_de_pago_previo pedidos_con_credito_ok pedido_rechazado

1.1.2 Diseño

El diseño de sistemas es el proceso de planificar, reemplazar o complementar un sistema organizacional existente. Establece cómo alcanzar el objetivo.

El diseño de un sistema produce los detalles que establecen la forma en la que el sistema cumplirá con los requerimientos identificados durante la fase de análisis.

El diseño de un sistema indica los datos de entrada, aquellos que serán calculados y los que deben ser almacenados. También se escribe a todo detalle los procedimientos necesarios para calcular datos individuales.

Se debe seleccionar las estructuras de archivos y los dispositivos de almacenamiento. Los procedimientos que se escriben indican cómo procesar los datos y producir las salidas.

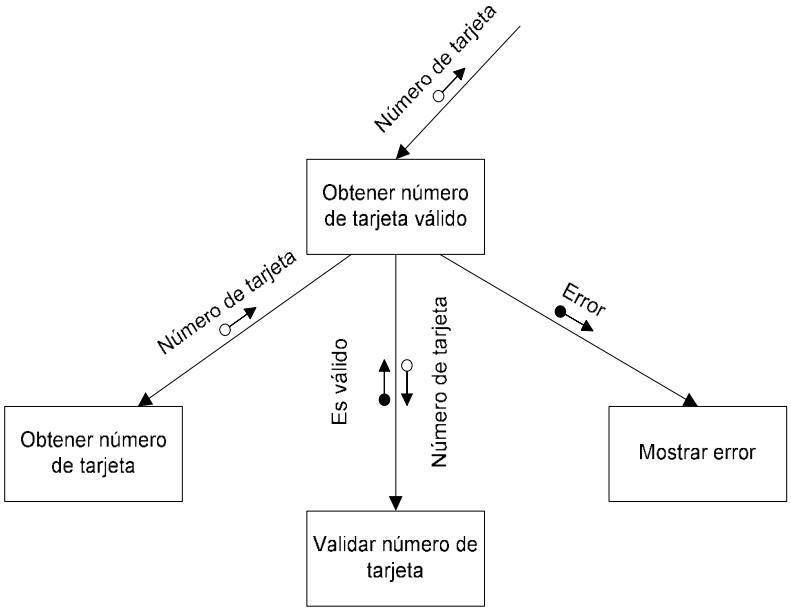
Al igual que en el análisis, en el diseño también se utilizan herramientas gráficas para poder dar a conocer las ideas que el diseñador desea que el equipo de desarrollo lleven a cabo.

1.1.2.1 Diagrama de estructura

El diagrama de estructura muestra la descomposición de un sistema en módulos, o cajas negras de las que se conoce sus entradas recibidas, salidas generadas y la función que lleva a cabo.

Un diagrama de estructura tiene la forma de un árbol y refleja la jerarquía de control de que módulos pueden invocar a otros módulos, al igual que los parámetros que pasan en las llamadas, esto se puede apreciar en la siguiente gráfica.

Figura 7. Ejemplo de un diagrama de estructura



1.1.2.2 Plantilla por interfaz función

La plantilla por internaza función permite leer de forma clara la interfaz de un módulo (parámetros de entrada y salida), la función del módulo y los elementos que utiliza (tablas, formulas, archivos).

Esta especificación permite definir un módulo sin entrar en excesivo detalle, lo cual da bastante libertad a los programadores para desarrollar la solución. A continuación se presenta un ejemplo de dicha plantilla.

Figura 8. Ejemplo de una plantilla por interfaz función

Modulo:	Validar número de tarjeta
Parámetro de entrada:	Número_de_tarjeta
Parámetro de salida:	Es_valido
Función del módulo:	Verifica que el número de tarjeta exista en la base de datos y por lo tanto sea válida.
Usa:	Utiliza la tabla tarjetas

1.1.2.3 Especificación por pseudo código

El pseudocódigo es un lenguaje de especificación de algoritmos que utiliza palabras reservadas y exige la tabulación de algunas líneas. Se realiza en un lenguaje informal muy parecido al lenguaje estructurado, el cual es más preciso y detallado que la especificación por interfaz función.

Tiene la ventaja de tener menor probabilidad de error en la codificación ya que no se especifica que hay qué hacer sino cómo hay que hacerlo. La siguiente gráfica muestra el pseudo código para el módulo que muestra los errores que se puedan dar en un programa.

Figura 9. Ejemplo de un pseudo código

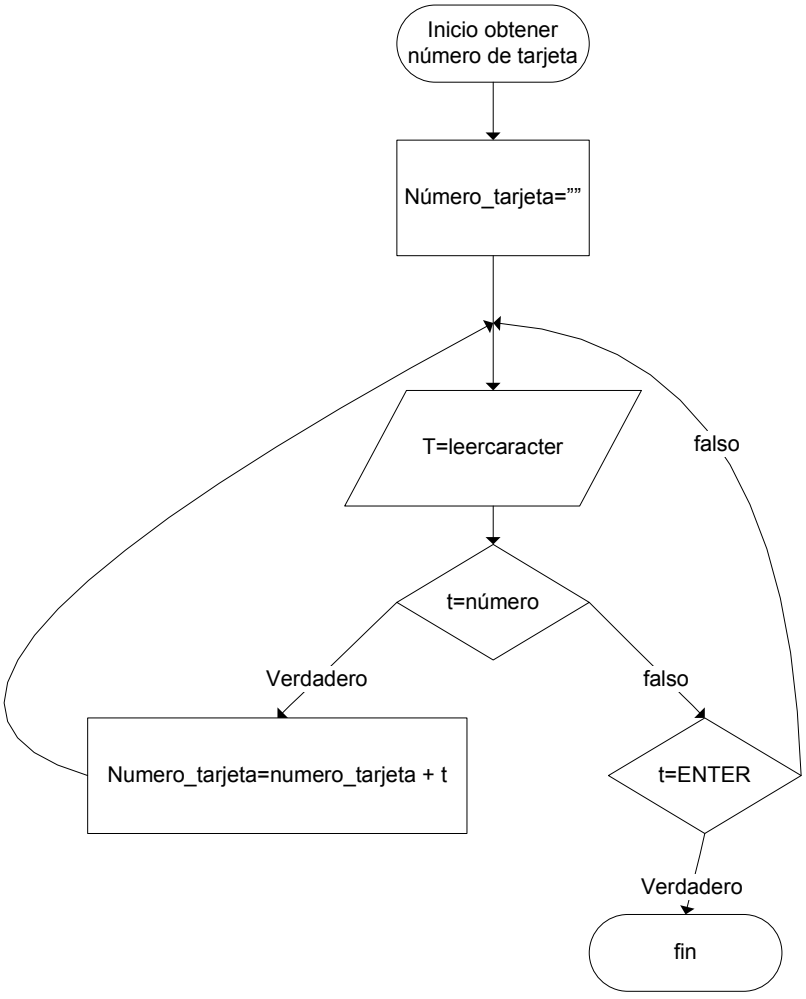
```
Modulo: Mostrar error
Parámetro de entrada: error
usa: tabla_de_errores
Inicio
        i=0;
        encontrar_error=falso;
        Mientras (no haya fin de tabla_de_errores ó encontrar_error=falso)
                si (tabla_de_errores[i].codigo=error)
                entonces
                        Imprimir(tabla_de_errores[i].mensaje)
                        encontrar_error=verdadero
                sino
                        i=i+1
                fin si
        fin mientras
        si (econtrar_error=falso)
        entonces
                imprimir ("Error no especificado")
        fin si
fin modulo
```

1.1.2.4 Diagramas de flujos

Los diagramas de flujo se utilizan tanto para la representación gráfica de las operaciones ejecutadas sobre los datos a través de todas las partes de un sistema de procesamiento de información, como para la representación de la secuencia de pasos necesarios para describir un procedimiento particular, diagrama de flujo detalle.

El diagrama de flujo utiliza unos símbolos normalizados, con los pasos del algoritmo escritos en el símbolo adecuado y los símbolos unidos por flechas, denominadas líneas de flujo que indican el orden en que los pasos debe ser ejecutados. Se puede apreciar en la siguiente gráfica un ejemplo de un diagrama de flujo

Figura 10. Ejemplo de diagrama de flujo



1.1.3 Codificación

Una vez se tenga la especificación de diseño, se da paso a la codificación del programa. El programador debe seguir con sumo cuidado el diseño que le ha sido entregado.

Ésta etapa no se dedica únicamente a crear el código fuente, también es necesario crear el manual técnico el cual tiene como objetivo permitir que cualquier programador pueda comprender el código fuente a su totalidad teniendo como material de apoyo éste documento y la especificación de diseño correspondiente a dicho módulo.

Se debe tener en esta documentación cómo y porque ciertos procedimientos se codificaron de determinada manera para poder realizar la pruebas y mantenimiento del programa.

1.1.4 Pruebas

Las pruebas son el factor crítico para determinar la calidad del *software* y tienen como objetivo descubrir algún tipo de error. Un caso de prueba es un conjunto de datos que el sistema procesará como entrada normal. Un caso de prueba es bueno cuando su ejecución conlleva una probabilidad elevada de encontrar un error.

Se recomienda que el esfuerzo para la verificación del *software* con utilice un 40% del tiempo total de desarrollo.

Existen diferentes niveles de prueba como lo son las pruebas parciales que se evalúan los programas que conforman un sistema, las pruebas de sistema que comprueban que la integración de cada modulo del sistema sea correcta. Pruebas de carga máxima para determinar si el sistema manejará el volumen de actividades proyectado, pruebas de almacenamiento, pruebas de tiempo de ejecución, pruebas de recuperación, pruebas de procedimientos entre otras, se pueden llegar a clasificar en dos, pruebas de código y pruebas de especificación.

1.1.4.1 Pruebas de código

Estas pruebas examinan la lógica del sistema, para seguir este método de prueba, el analista desarrolla casos de prueba que produzcan la ejecución de cada instrucción en el programa o módulo.

Se debe tener en cuenta que no todos los errores de *software* se pueden descubrir verificando todas las rutas de un programa. Esta prueba no determina si se cumple con los requerimientos del sistema y tampoco verifica el rango de los datos que aceptará el programa.

1.1.4.2 Pruebas de especificación

El analista debe examinar las especificaciones que señalan lo que el programa debe hacer y cómo lo debe llevar acabo bajo diferentes condiciones, éstas son las pruebas de especificación.

Después de desarrollar los casos de prueba para cada condición o combinación de condiciones, se mandan para su procesamiento. Por medio de un estudio de resultados, se puede determinar si el programa cumple o no con las especificaciones.

1.1.5 Mantenimiento

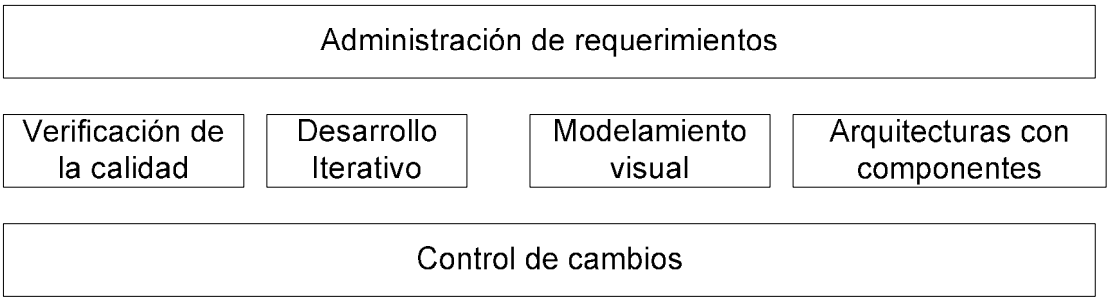
Una vez el producto está terminado se debe entregar al usuario final, Dependiendo el tipo de licencia con el cual el *software* es distribuido, se obliga a los creadores a darle mantenimiento preventivo y correctivo a dicho producto.

1.2 Ciclo de vida iterativo incremental

El proceso Unificado es un proceso de desarrollo de *software* configurable que se adapta a proyectos que varían en tamaño y complejidad. Se basa en muchos años de experiencia en el uso de la tecnología de objetos en el desarrollo de *software* de misión crítica en una variedad de industrias.

El Proceso Unificado describe como utilizar de forma efectiva procedimientos comerciales probados en el desarrollo de *software* para equipos de desarrollo de *software*, conocidos como “mejores prácticas”, las cuales son mencionadas en la siguiente gráfica..

Figura 11. Mejores prácticas del desarrollo de *software*



1.2.1 Desarrollo iterativo

Dados los sistemas de *software* tan complejos de la actualidad, no es posible hacer de manera secuencial la definición completa del problema al que se afronta, diseñar la solución completa y construir el *software* y por último probarlo. Al trabajar de esta forma se puede llegar al descubrimiento de defectos en fases posteriores al diseño que pueden aumentar los costos. Este es un riesgo demasiado alto ya que puede llevar a la cancelación del proyecto, es por esta razón que se deben dar refinamientos sucesivos para permitir un entendimiento incremental del problema.

Esto se pude lograr con reuniones periódicas con los usuarios para poder tener retroalimentación. El progreso es medido conforme avanzan las implementaciones. Estas metas permiten tener al equipo de desarrolladores mantenga la atención en producir resultados.

1.2.2 Administración de requerimientos

El proceso unificado lleva un registro y documentación detallada de los cambios y decisiones que se han tomado a lo largo del proyecto. Los requerimientos del negocio son fácilmente obtenibles al utilizar los casos de uso.

También permite documentar la funcionalidad y las restricciones requeridas que han sido producto de las entrevistas con los usuarios.

1.2.3 Arquitectura basada en componentes

Se enfoca en el desarrollo de una arquitectura ejecutable y robusta. Estos componentes deben ser tan específicos que permitan el uso más efectivo del *software* y sean intuitivamente comprensibles. Estos componentes deben ser derivados de un caso de uso más importante.

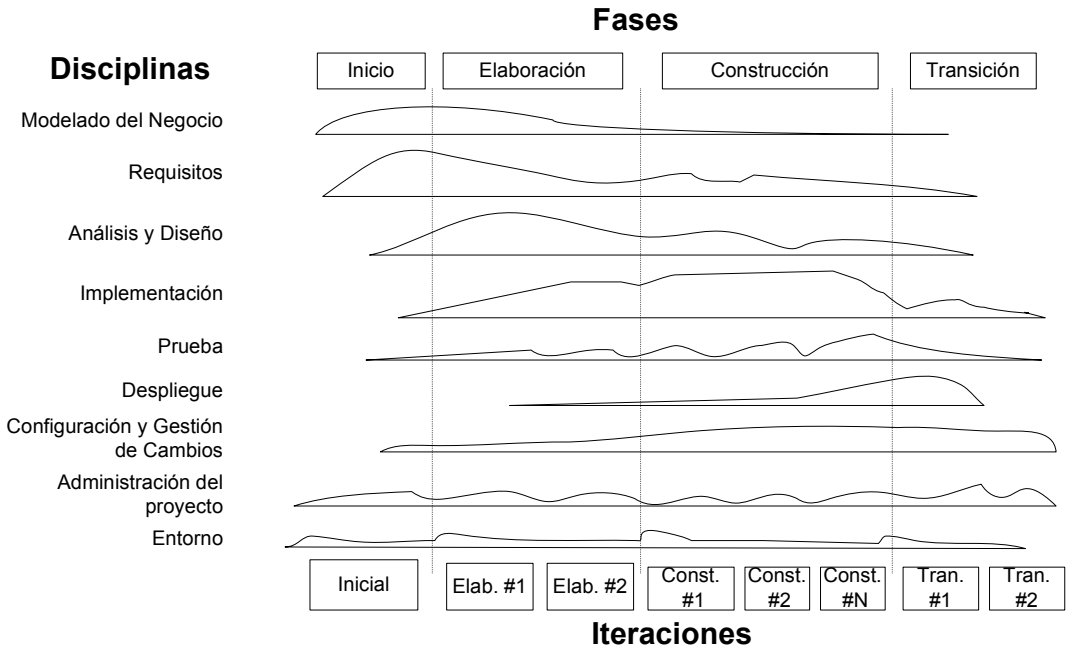
1.2.4 Modelación visual

Uno de los componentes clave es el UML. UML es acrónimo de *Unified Modeling Language* que es un lenguaje de modelación visual para modelar un sistema. Esta permite capturar la estructura y comportamientos de las arquitecturas y los componentes. Este lenguaje de modelado evita la comunicación ambigua y es tan claro que mantiene la consistencia entre un diseño y su implementación, además muestra como encajan de forma conjunta los elementos del sistema.

1.2.5 Modelo del ciclo

Al igual que el ciclo de vida clásico, se puede representar un modelo para el ciclo de vida del proceso unificado.

Figura 12. Ciclo de vida del proceso unificado



En la parte superior de la gráfica se puede apreciar cuatro fases en las que se divide el ciclo de vida a lo largo del tiempo. En la parte inferior las diferentes iteraciones que se realizan, pueden existir diferentes iteraciones para una misma fase, no se puede avanzar a menos que la iteración anterior este completa.

Al lado izquierdo se enumera las diferentes disciplinas que se deben realizar. El área debajo de la curva de cada disciplina representa el esfuerzo que se debe hacer, el cual varía en cada fase e instante de la iteración.

1.2.6 Las 4 fases

Esta metodología se puede dividir en el tiempo en cuatro fases, las cuales representan el tipo de trabajo que se está haciendo sobre el proyecto.

- Inicio.
- Elaboración.
- Construcción.
- Transición.

1.2.6.1 Inicio

La fase de Inicio pretende responder a preguntas como ¿Cuál es el objetivo del proyecto?, ¿Es factible llevarlo a cabo?, ¿Es mejor construir el producto o comprarlo?, ¿Cuánto va a costar? Esto tiene como objetivo explorar el problema antes de decidir a abordarlo. Ayuda a obtener una idea global de la forma en que funciona el sistema.

El propósito de esta fase no es crear un plan preciso y fiable o especificar todos los requisitos, para esto se tiene otras iteraciones, las cuales ayudarán a especificar lo que se ha querido modelar.

Es una fase breve que no debe tardar más de unas pocas semanas, se debe tener claro que la investigación en profundidad se deja para la fase de elaboración.

Los productos no se terminan, se hace un alrededor de un 20%, lo cual es necesario para entender el funcionamiento del negocio y los requisitos que puede llegar a necesitar.

1.2.6.2 Elaboración

En esta fase se analiza el problema a fondo, con una claridad suficiente para realizar un diseño claro que ayude a resolver el problema planteado.

Se establece cimientos de la arquitectura el cual permitirá crear un producto de *software* de calidad. Se demuestra que la arquitectura propuesta soportará la visión con un coste razonable y en un tiempo razonable.

Al inicio de cada iteración se establece los riesgos para poder eliminarlos. La identificación de de los riesgos en estas fases ayudará a establecer planes de contingencia que evitarán retardos en la entrega del proyecto.

Se han establecido los casos de uso críticos de los cuales depende el éxito del proyecto.

Una vez terminada esta fase, se tiene un prototipo ejecutable del sistema, el cual se le puede entregar al cliente para obtener retroalimentación y de esta forma mejorar los cambios que el cliente desee en una fase temprana de desarrollo.

La visión del producto es estable y se sabe que este prototipo evolucionará hasta convertirse en el sistema final el cual ya ha sido probado en parte por el cliente.

Es el punto de no retorno, al proyecto no se le podrá dar marcha atrás.

1.2.6.3 Construcción

En esta fase se debe alcanzar la capacidad operacional del producto de forma incremental a través de las sucesivas iteraciones.

En esta fase se tiene los modelos completos (Casos de uso, análisis, diseño, despliegue e implementación).

Se debe tener la versión beta, la cual es útil para identificar problemas en el *software* que no hayan sido encontradas en la pruebas.

Todos los componentes, características y requisitos, que no lo hayan sido hechos hasta ahora, han de ser implementados, integrados y testeados para poder asegurase que el cliente reciba lo que ha pedido.

Se elabora el manual inicial de usuario que es proporcionado con la versión beta.

1.2.6.4 Transición

Se prueba la versión beta de cara al usuario y con testers contratados. Con esta información se desarrollar nuevas versiones actualizadas del producto con menos errores y mayor porcentaje de aceptación por el cliente.

En esta fase se pone el producto en manos de los usuarios finales, el cual se obtiene gracias a la información obtenida por la versión beta.

Se contratan documentadores para completar los manuales que se entregarán al usuario final y los pueda utilizar como referencia en cualquier momento.

En general se realizan tareas relacionadas con el ajuste, configuración, instalación y usabilidad del producto, también se capacitan a los usuarios.

Si se logra tener un usuario sea autosuficiente se puede decir que se ha tenido un éxito al crear un *software* fácil de usar y gran calidad.

Se puede decir que se tiene un producto final que cumpla los requisitos esperados, que funcione y satisfaga suficientemente al usuario.

1.2.7 Los flujos de trabajo

Los flujos de trabajo se llevan en forma paralela y el esfuerzo requerido para cada uno de ellos depende de la fase en la que se encuentre. Los flujos del Proceso Unificado son:

- Modelado del negocio.
- Requisitos.
- Análisis y diseño.
- Implementación.
- Prueba.
- Despliegue.
- Configuración y gestión de cambios.
- Administración del proyecto.
- Entorno.

1.2.7.1 Modelado del negocio

Los propósitos del flujo del modelado del negocio son:

- Entender la estructura y las dinámicas de la organización en donde se va a desarrollar el sistema (empresa objetivo).

- Comprender los problemas de la empresa objetivo e identificar potenciales mejora.

- Asegurar que los clientes, los usuarios y los desarrolladores tengan una comprensión común de la empresa objetivo.

- Derivar que los requerimientos del sistema que ayudarán la empresa objetiva.

Al inicio se evalúa el estatus actual de la empresa en donde se va a desplegar la aplicación. Deben quedar claros los procesos actuales, las herramientas, las competencias de las personas, los clientes, las necesidades técnicas y las áreas de mejora.

Se define un vocabulario común que se puede utilizar en todas las descripciones textuales del negocio, especialmente en la descripción de los casos de uso del negocio. También se necesita identificar las reglas del negocio, éstas se pueden obtener leyes y regulaciones, otras peden ser estándares de la empresa, algunas reglas del negocio expresan los objetivos de lo que se desea hacer con el modelado del negocio. Luego se debe expresar las reglas utilizando un lenguaje formal.

Ya que se tiene claro en entorno donde se va a realizar la aplicación se definen límites claros para no exceder el modelado del negocio, se plantea una visión de la futura empresa objetivo. Es necesario establecer objetivos primarios de la empresa objetivo.

El modelado del negocio genera los siguientes artefactos: Especificación suplementaria del negocio, la visión de la empresa y el glosario.

1.2.7.2 Requisitos

El establecimiento de requisitos ayuda a establecer y mantener un acuerdo entre el cliente y cualquier otro *stakeholder* en lo que el sistema debe hacer el producto. En esta etapa se ayuda a los desarrolladores a tener una mejor comprensión de los requerimientos del sistema.

No se puede seguir adelante en el proyecto si no se termina de delimitar el sistema. Una vez hecho esto se tiene las bases para la planeación del contenido técnico por cada iteración y las bases para estimar los costos y tiempos requeridos para realizar el proyecto. Estas estimaciones no se pueden realizar sin haber definido el lenguaje de programación que pueda satisfacer los requerimientos del cliente.

En este flujo se estableceremos qué hace exactamente el sistema, estos requerimientos son el contrato que garantiza el tipo de trabajo que se llevará a cabo.

Este flujo permite identificar dos tipos de requisitos, aquellos que se pueden modelar con un caso de uso, o sea los funcionales, y aquellos de especificaciones adicionales o no funcionales.

Estos requisitos se obtienen al recopilar hechos de todos los interesados por medio de entrevistas, encuestas y/o observación por mencionar algunas técnicas de recopilación de datos.

En esta etapa se lleva a cabo el diseño de la interfaz que se propone sea utilizada por el usuario, el cliente la evalúa y da su aprobación si considera que satisface sus necesidades.

1.2.7.3 Análisis y diseño

Tiene como objetivo traducir los requisitos a una especificación que describe cómo implementar el sistema.

El análisis especifica con mayor detalle qué hace el sistema mientras que el diseño crea informes de cómo se debe realizar el proyecto.

Se debe tener en cuenta que no es necesario obtener un modelo de análisis independiente ya que esta apoyado en el modelado de requisitos.

El modelo de diseño es importante ya que sin el no se puede codificar el sistema, no se debe olvidar que el diseño debe ser suficientemente claro para que el sistema pueda ser implementado sin ambigüedades.

Es posible que la implementación se haga automáticamente dependiendo del tipo de herramienta C.A.S.E. con la que se este trabajado.

La documentación de la arquitectura *software* captura varias visiones arquitectónicas del sistema, se debe tener cuidado ya que una mala manipulación de versiones puede causar serios problemas en el desarrollo del sistema.

1.2.7.4 Implementación

En esta disciplina se implementan las clases y objetos en ficheros fuente, binarios, ejecutables y demás objetos del sistema.

Se debe definir un plan de integración para asegurarse que exista una transparencia en la unión de los diferentes módulos. Este plan contempla qué se va a implementar, en qué orden se va a integrar y se asegura que la integración sea de forma incremental.

La estructura de todos los elementos implementados forma el modelo de implementación.

Cada implementador debe probar cada unidad que produzca antes de las pruebas de integración para asegurase que el producto que esta terminando sea funcional.

Es recomendable que en las primeras fases se desarrollen prototipos para que el cliente tenga una mejor idea de la visión que tiene el equipo de desarrollo del trabajo que se ha pedido.

1.2.7.5 Prueba

En este flujo se evaluar la calidad del producto, se debe tener presente que no es una actividad final para aceptar o rechazar el producto.

Para evitar retardos se debe llevar cabo durante todo el ciclo de vida para poder asegurarse que se está cumpliendo con las metas al crear un producto de gran calidad.

Las pruebas se enfocan principalmente en evaluar o fijar la calidad del producto por medio de diferentes prácticas.

Se debe buscar y documentar los defectos de la calidad del *software*. Se debe probar la validez de las suposiciones hechas en la especificación del diseño y especificación de requerimientos por medio de demostraciones concretas.

En la etapa de prueba se debe buscar y exponer las debilidades del producto de *software*, para que este esfuerzo sea exitoso, se debe tener un enfoque negativo y destructivo en vez del enfoque constructivo de las otras disciplinas.

1.2.7.6 Despliegue

Una vez terminado el producto, se establece el plan de despliegue, para lo cual se debe tomar en cuenta el tiempo de capacitación, tiempo de instalación, tiempo de validación del programa entre otras cosas.

Se elaboran los diferentes materiales de soporte como lo son los tutoriales que ayudarán a los usuarios a obtener el máximo provecho del *software* y el material de entrenamiento que es útil en las capacitaciones.

Se selecciona un método de distribución acorde a mercado del producto y los objetivos de la gerencia, esta puede ser por medio de diferentes puntos de venta o un sitio en Internet.

Esta fase incluye la formación usuarios y vendedores.

Se desarrolla con mayor intensidad en la fase de transición, pero debe empezar en fases anteriores.

1.2.7.7 Configuración y gestión de cambios

El proceso unificado genera gran variedad de artefactos en sus diferentes disciplinas, estos artefactos describen partes de un sistema que por naturaleza evoluciona de una forma dinámica. Estos cambios deben quedar bien documentados para poder mantener la integridad de todos los artefactos que se crean durante el proceso de desarrollo del producto.

La creación de diferentes prototipos y la aceptación de ciertos modelos por parte del cliente deben quedar claramente documentados. Es importante mantener información del proceso evolutivo que han seguido y de esta forma evitar problemas de integración.

Un paso muy importante en este flujo es establecer y realizar una planificación para el proyecto de configuración y control de cambios. Aquí se debe establecer las políticas para administración del proyecto. También se establecen las políticas y los procesos para controlar los cambios del producto.

1.2.7.8 Administración del proyecto

Es el arte de balancear objetivos competentes, administrar riesgos y superar limitantes para poder entregar de forma exitosa un producto que satisfaga tanto a los clientes como a los usuarios.

Su propósito es el de proveer un marco de trabajo para administrar el proyecto de *software* y los riesgos. También provee una guía práctica para planificar, ejecutar y monitorear el proyecto.

Esta disciplina no toma todos los aspectos de la administración de proyectos como lo es la gestión de personal (contratación, capacitación), gestión de presupuesto, gestión de contratos con clientes y proveedores.

Esta disciplina se enfoca principalmente en los aspectos importantes del proceso de desarrollo iterativo como lo son la gestión de riesgos, la planificación de un proyecto iterativo por medio del ciclo de vida para una iteración en particular y el monitoreo del progreso de un proyecto iterativo.

En la iteración inicial en la fase de inicio, la disciplina de administración del proyecto inicia con la concepción de un nuevo proyecto, durante la cual se crea y revisa la visión inicial, la factibilidad del negocio y el listado de riesgos. Todo esto con el objetivo de obtener los fundamentos necesarios para proceder con el proyecto.

Se crea un plan de desarrollo de *software* básico en el cual se crea el plan de iteración inicial. En esta iteración inicial se evalúa los limites del proyecto y sus riesgos para completar los artefactos de visión, listado de riegos y *Business Case*.

Antes de cambiar de iteración en el proyecto, se hace una planificación de la siguiente iteración en la cual se establecen los objetivos e hitos que debe cumplir. Si la iteración es mayor de seis meses, se recomienda hacer una revisión a media iteración.

La planificación de la siguiente versión debe ser detallada, también se recomienda documentar las lecciones aprendidas y actualizar el plan del proyecto para las futuras iteraciones.

1.2.7.9 Entorno

Esta disciplina se enfoca en las actividades necesarias para configurar el proceso para un proyecto. Describe las actividades requeridas para el desarrollo de las guías que soporten un proyecto. El propósito de las actividades de entorno es de proveer a la organización de desarrollo de *software* con el ambiente de desarrollo tanto de proceso como de herramientas que van a soportar al equipo de desarrollo.

Las responsabilidades que incluye son:

- Selección y adquisición de herramientas.
- Establecer y configurar las herramientas para que se ajusten a la organización.
- Configuración del proceso.
- Mejora del proceso.
- Servicios técnicos.

2 PROGRAMACIÓN EXTREMA

La programación extrema (XP) es una disciplina de desarrollo de *software*, que se enfoca en satisfacer las necesidades del cliente. Esta metodología esta diseñada para entregar el *software* al cliente cuando lo necesita y faculta a los desarrolladores a responder a los cambios del cliente sin importar en que parte del desarrollo vaya el proyecto.

Esta metodología, también apoya el trabajo en equipo, administradores, desarrolladores y el cliente son parte de un equipo dedicado a crear un *software* de gran calidad.

La palabra “extrema” en el nombre de esta metodología proviene de tomar principios y prácticas de sentido común a niveles extremos.

- Se debe revisar el código todo el tiempo por medio de la programación en parejas, ya que las revisiones de código son buenas.

- Ya que las pruebas ayudan a mejorar la calidad, se debe probar todo el tiempo con las pruebas unitarias o con la prueba funcional que hace el cliente.

- Si las pruebas de integración son importantes, entonces se debe hacer una integración continua para poder realizar las pruebas.

- Un buen diseño ayuda a la calidad de un producto por lo que se debe hacer esto todos los días por medio de la recodificación.

- Es más fácil modificar algo simple, por lo que hay que hacer las cosas lo más sencillamente posible que funcionalmente sea aceptable.

- Ya que las iteraciones son buenas, se debe iterar lo más pronto posible y no esperar meses.

La metodología XP mejora un proyecto de *software* al mejorar cuatro valores.

- **Comunicación**. Al estar en contracto con el cliente y con los compañeros de desarrollo.

- **Sencillez**. Tratando de mantener el diseño simple y claro.

- **Retroalimentación**. Que se obtiene al probar el *software* desde el primer día y por la involucración del .cliente en el desarrollo del proyecto.

- **Coraje**. Para afrontar los cambios de tecnología y requerimientos que se presenten en el desarrollo del proyecto.

Estos valores se deben nutrir para poder sacar adelante un proyecto de desarrollo de *software* ya que gracias a ellos se puede cumplir con los siguientes principios:

- **Retroalimentación rápida**. La única forma de saber si algo se hace bien o mal es por medio de una retroalimentación, entre más rápida se tenga esta información, más rápido se podrá tomar medidas correctivas. Si se tiene una comunicación clara, se podrá interpretar de forma correcta lo sucedido y se podrán realizar los cambios necesarios para no afectar el proyecto.

- **Asumir la sencillez**. Entre más sencilla se hagan las cosas, más fácil será entenderlas o recordarse de lo hecho. El tiempo utilizado para las nuevas versiones se debe utilizar para crear algo nuevo y no para comprender algo existente.

- **Lograr cambios incrementales**. Todo cambio debe agregar valor al proyecto de *software*, no se deben realizar grandes cambios que, al final del día, no permitan mejorar lo que ya se tiene.

- **Aceptar el cambio**. Si no se está conciente que los cambios son la única constante al desarrollar un proyecto, se puede llegar a crear un ambiente de desarrollo inadecuado, que no permite generar nuevas ideas para mejorar el proyecto.

- **Trabajar con claridad**. Hay que estar conciente que se está trabajando en equipo, si no se trabaja con claridad va ser muy difícil que otro compañero pueda entender lo que ya se ha desarrollado y se pierdan recursos comprendiendo o rehaciendo algo que ya está.

Los valores y principios ya mencionados permitirán que las prácticas de la metodología XP permitan crear un *software* de mayor calidad que satisfaga las necesidades del cliente en el momento que él desee.

Otros principios que ayudarán al éxito del proyecto son:

- Enseñar a aprender.
- Jugar a ganar.
- Comunicación abierta y honesta.
- Trabajar con los instintos de las personas, no contra ellos.
- Aceptar la responsabilidad.

2.1 Las instalaciones

Para poder hacer algo es necesario tener el lugar adecuado para hacerlo. La programación extrema necesita de mucho espacio en común.

Hay que tener en cuenta que esta metodología es una disciplina comunal de desarrollo de *software*. Los miembros del equipo necesitan poder verse unos a otros, y aceptar que otros van a escuchar los problemas personales que puedan surgir dentro del grupo de desarrollo.

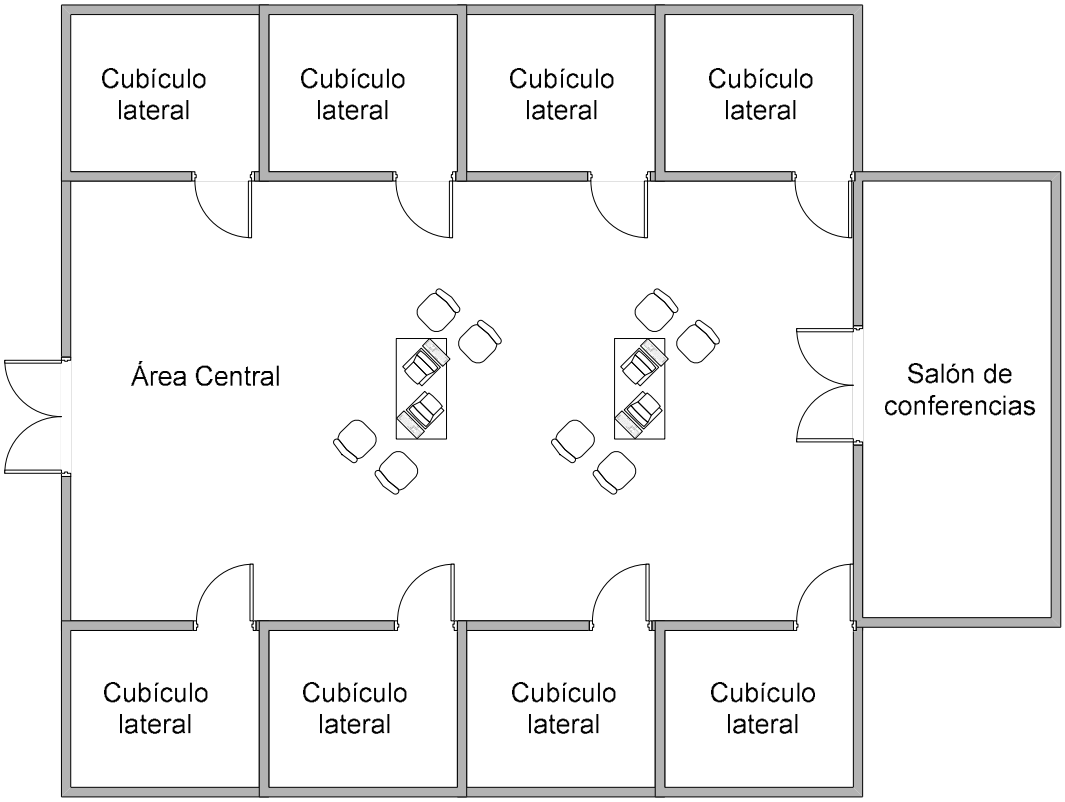
La estructura recomendada para un equipo de programación extrema es:

- Tener un espacio central común.
- Tener pequeños cubículos en los laterales de dicho espacio.
- Colocar las mejores máquinas en el espacio central común.

Los miembros del equipo pueden tener sus efectos personales en los cubículos y pueden ir a ellos cuando no quieran ser interrumpidos.

Al tener las máquinas en el área central común, se obligará a una persona que desee codificar, a ir dicho sector, de tal forma que todo el mundo pueda ver lo que está haciendo. Las parejas se pueden formar fácilmente y cada pareja puede atraer la energía de las otras parejas que también están desarrollando al mismo tiempo.

Figura 13. Instalación ideal para un equipo extremo



Si el equipo debe trabajar en el edificio del cliente, hay que tener cuidado, ya que lo primero es la comodidad del equipo de programación extrema. Ellos deben tener la luz adecuada, el mobiliario adecuado, un movimiento fluido, un ambiente agradable que permita el desarrollo de la creatividad.

Tomar el control del entorno físico envía un mensaje convincente al equipo de no permitir que intereses opuestos irracionales en la organización impidan conseguir el éxito.

2.2 Planificación

Se determina el ámbito de la siguiente versión, combinando las prioridades del negocio y las estimaciones técnicas. El desarrollo del *software* es siempre un dialogo que evoluciona entre lo posible y lo deseable.

El cliente debe tener claro que es imprescindible y que su presencia no es opcional, el cliente debe establecer una prioridad ya que únicamente él es el que conoce lo que tiene más valor para el negocio.

El cliente establece una propuesta del contendido de la versión, la cual es discutida con el equipo de desarrollo para poder establecer la verdadera urgencia de cada parte del proyecto propuesta para dicha versión y la fecha de entrega.

Una vez se ha establecido lo necesario para la versión, se realiza una programación detallada divida en iteraciones. Los programadores necesitan la libertad de realizar las partes con mayor riesgo primero para reducir el riesgo global del proyecto.

Se deben realizar versiones pequeñas que contengan los requisitos más importantes para el negocio. Éstas versiones son muchas historia juntas que hacen que el negocio tenga sentido. Dicha versión debe ser atómica por lo que no se puede implementar a medias y luego presentarla como una versión completa. Se debe recordar que entre más pequeña es la versión, más rápido es evaluada por el cliente y se tiene una pronta retroalimentación.

2.2.1 Planificación de la versión

La versión se planifica con un dialogo entre el cliente y el equipo de desarrollo. Este dialogo se debe dar en un ambiente de confianza mutua y respeto a la contraparte. Tanto el cliente como el equipo de desarrollo deben permitir que su contraparte realice su trabajo.

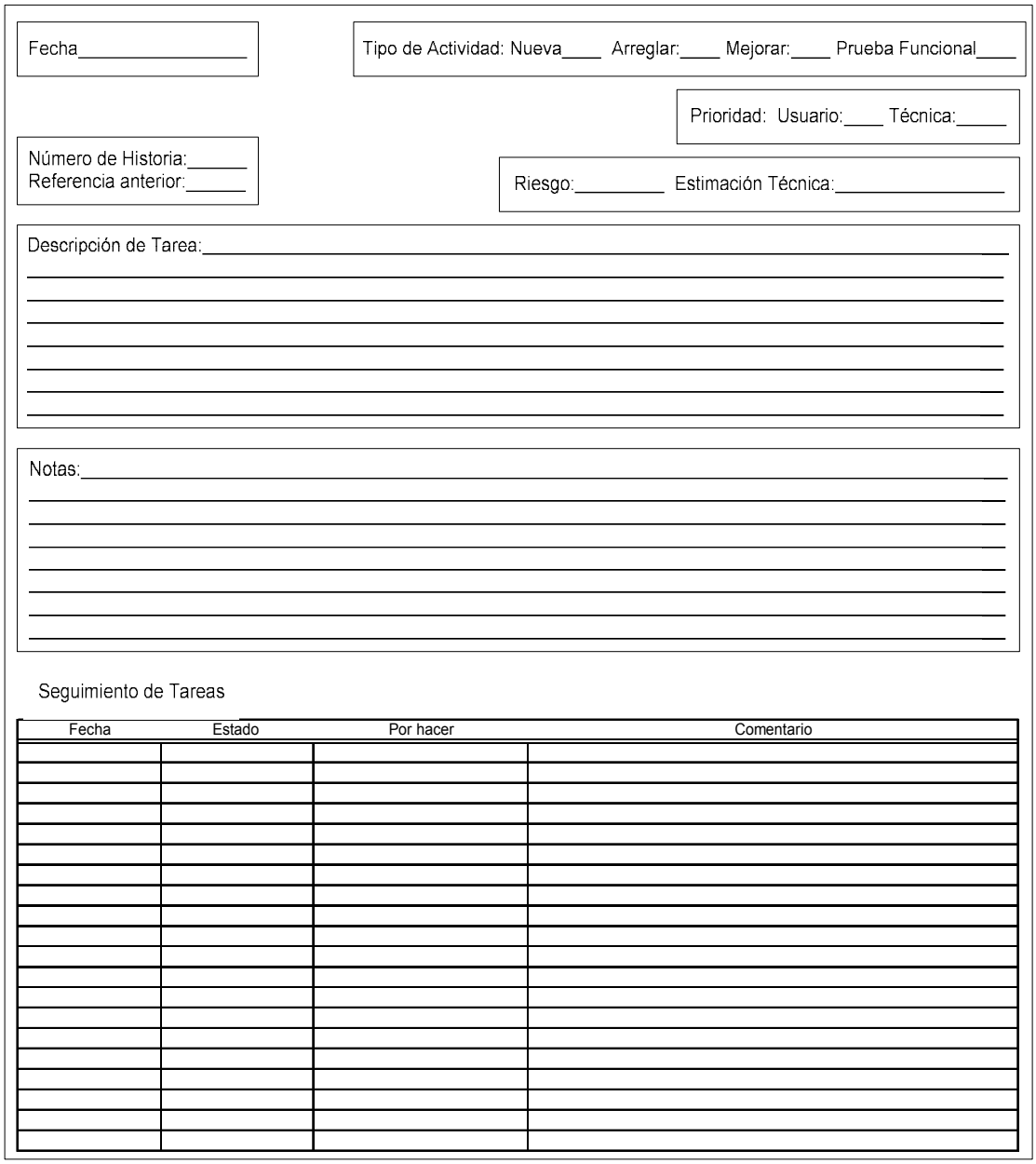
Hay que enfatizar que el objetivo de esta planificación es de maximizar el valor del *software* producido por el equipo. Del valor del *software*, se debe descontar el coste de su desarrollo y el riesgo contraído durante el desarrollo.

Se obtiene una mejor planificación si se tiene en cuenta que hay que invertir lo menos posible para poner la mayor funcionabilidad válida en producción tan rápidamente como sea posible.

En la planificación de versión se utilizan fichas de historias, las cuales maneja la siguiente información:

- Fecha.
- Tipo de actividad (nueva, arreglar, agregar).
- Numero de historia.
- Prioridad (para el cliente y para el equipo de desarrollo).
- Descripción de la historia.
- Notas extras.
- Bitácora de labores (fecha, estado, por hacer, comentario).

Figura 14. Ejemplo de una ficha de historia



La planificación de versión consta de tres fases, las cuales se describen a continuación.

2.2.1.1 Fase de exploración

Tiene como propósito dar a conocer una estimación de lo que el sistema realizará. Se puede dividir esta fase en tres pasos:

- **Escribir una historia**. El cliente escribe una historia en la ficha ya mencionada, se incluye el nombre y un párrafo que explique el objetivo de la historia.

- **Estimar una historia**. El equipo de desarrollo calcula el tiempo que le llevará implementar dicha historia, si no se puede estimar la historia, se le pide al cliente que aclare o divida la historia. Los tiempos se estiman con el tiempo ideal de ingeniería (TII). Antes de comprometerse a una programación, se mide la relación entre el tiempo ideal y el calculado.

- **Dividir una historia**. Si el equipo de desarrollo no puede estimar una historia global, o si el cliente se da cuenta qué parte de una historia es más importante que el resto, el cliente puede dividir la historia en dos o más historias.

2.2.1.2 Fase de compromiso

Tiene como propósito que el cliente establezca el ámbito y la fecha de la próxima versión y el equipo se comprometa a entregarla. Se divide en cuatro fases:

- **Clasificar por valor**. El cliente tiene la obligación de clasificar las historias entre aquellas que son cruciales para el sistema, las que son menos importantes y las que le agradaría tener.
- **Clasificar por riesgo**. El equipo de desarrollo clasifica las historias en aquellas que puede estimar con precisión, aquellas que se pueden estimar razonablemente bien y aquellas que no pueda estimar.

- **Establecer velocidad**. El equipo de desarrollo le dice al cliente la rapidez con que el equipo pude trabajar en el TII por mes o semana.

- **Escoger el ámbito**. El cliente escoge el conjunto de fichas de la versión, bien poniendo o estableciendo una fecha de finalización y escogiendo las fichas sobre la base de sus estimaciones y la velocidad del proyecto o bien escogiendo las fichas y calculando la fecha.

2.2.1.3 Fase de dirección

En esta fase es en donde se actualiza el plan basándose en lo que han aprendido tanto el cliente como el equipo de desarrollo.

- **Iteración**. Al comienzo de cada iteración, el cliente escoge la historia más valiosa a ser implementada en una iteración. Las historias para la primera iteración deben producir un sistema que funciona de punta a punta.

- **Recuperación**. Si el equipo de desarrollo se da cuenta que ha sobreestimado su capacidad creación de código, puede preguntarle al cliente qué grupo de historias son más valiosas para mantener en la versión actual basándose en la nueva velocidad y estimaciones.

- **Nueva historia**. Es muy probable que el cliente desee agregar una nueva historia a mitad del desarrollo de una versión, se agrega una nueva ficha, el equipo de desarrollo estima la historia y el cliente elimina en el resto del plan de historias que tengan una estimación equivalente y se agrega la nueva historia a la versión.

- **Volver a estimar**. Si el equipo de desarrollo se da cuenta que el plan no proporciona un panorama claro para el desarrollo, puede volver a estimar el resto de historias que quedan y la velocidad.

2.2.2 Planificación de la iteración

La planificación de la iteración es muy parecida a la planificación de la versión. La primera modificación se puede apreciar con las personas involucradas ya que son únicamente los programadores individuales los que afectan esta planificación.

Otra variante es que se utilizan fichas de tarea en vez de fichas de historias. Se sigue teniendo tres fases identificadas con los mismos nombres pero las acciones de cada una cambian.

Figura 15. Ejemplo de ficha de tarea

Fecha_____

Número de Historia:_____

Ingeniería de Software: ____ Estimado de la tarea:_____

Descripción de Tarea:_____

Notas de los ingenieros de software:_____

Seguimiento de Tareas

Fecha	Estado	Por hacer	Comentario

2.2.2.1 Fase de exploración

- **Escribir una tarea**. Se selecciona una historia para la iteración y el programador la transforma en tareas. Esta tarea es más pequeña que la historia y puede que existan algunas que no estén directamente relacionadas con alguna historia particular.

- **Dividir una tarea / combinar tareas**. Si no se puede estimar una tarea en unos pocos días, se divide en tareas más pequeñas, por lo contrario si varias tareas necesitan unas horas, hay que combinarlas para formar una tarea mayor.

2.2.2.2 Fase de compromiso

- **Acepta una tarea**. Una vez que se tienen las tareas, es responsabilidad del programador el seleccionar una tarea y comprometerse a terminarla.

- **Estimar una tarea**. El programador estima el TII para cada tarea. Esto esta condicionado a obtener ayuda de otro programador que puede estar más familiarizado con el código a elaborar.

- **Conjunto de factores de carga**. Cada miembro del equipo de desarrollo elige su factor de carga para cada iteración, éste número indica el tiempo que dedicará realmente al desarrollo. Este número no debe ser muy alto ya que representaría que no va a dedicar tiempo a ayudar a sus compañeros.

- **Balanceo**. Cada programador suma sus estimaciones de las tareas y las multiplica por su factor de carga. Los programadores que se comprometen en exceso deben ceder algunas tareas, si todo el equipo está comprometido en exceso, se debe encontrar la forma de volver al equilibrio.

2.2.2.3 Fase de dirección

- **Implementar una tarea**. Un programador coge una ficha de tareas, encuentra un compañero, escriben los casos de prueba para la tarea, realizan todo el trabajo, lo integran y generan nuevo código cuando el juego de pruebas globales funciona.

- **Registrar el progreso**. Cada tres o dos días, un miembro del equipo pregunta a cada programador cuánto tiempo ha utilizado a cada una de sus tareas y cuanto le hace falta.

- **Recuperación**. Si un programador está comprometido en exceso, está obligado a pedir ayuda y evitar reducir el ámbito de algunas tareas, pedir al cliente que reduzca el ámbito de algunas historias, quitar tareas no esenciales, obtener una mejor ayuda o pedir al cliente que retrase algunas historias a una iteración posterior.

- **Verificar la historia**. Cuando ya se tiene las pruebas funcionales preparadas y completadas las tareas para una historia, se ponen en funcionamiento las pruebas funcionales para verificar que la historia funciona. Los casos interesantes que surjan durante la implementaron pueden añadirse al conjunto de pruebas funcionales.

2.3 Diseño

Cuando se elabora el diseño hay que recordar que es más fácil dar a entender un diseño sencillo que uno complicado, por lo que hay que crear estrategias de diseño que propongan el diseño más sencillo posible, y que tenga relación con el resto de los objetivos. Estos diseños deben ser bien claros para que las personas puedan entenderlos con bastante claridad sin tener que perder mucho tiempo tratando de entender lo quiere decir.

El diseño debe encontrar de una forma rápida una técnica para comprobarse, esto se logra consultando con los otros miembros del equipo si le parece el diseño y escuchar los cambios propuestos. El bueno retroalimentarse de lo que se aprende en el diseño.

Crear el diseño, probarlo, mejorarlo y aprender de él, es un proceso que se debe hacer de una forma cíclica y lo más pronto posible con tal de mejorar la calidad del diseño.

El éxito del diseño de la metodología de programación extrema se basa en asumir la simplicidad, ya que como los proyectos están sujetos a cambios es muy probable que el tiempo invertido en un diseño complejo sea tirado a la basura.

Es necesario tener la mentalidad del cambio incremental, se debe diseñar un poco cada vez, hay que estar concientes que el sistema nunca estará completamente diseñado, ya que como hay cambios, existirán partes del sistema que nunca cambiarán y otras que evolucionarán a lo largo del tiempo.

Ya concientes que el diseño irá cambiando a lo largo del tiempo, entonces se debe enfocar el esfuerzo en lo que se desea hacer. El diseño debería de ser suficientemente completo para adaptarlo a los objetivos actúales pero no más.

Una estrategia de diseño recomendada por el creador de la programación extrema es la siguiente:

1. “Comenzar con una prueba, así sabremos lo que estamos haciendo. Tenemos que hacer una cierta cantidad de diseño mínimo para poder escribir la prueba: ¿Cuáles son los objetivos y sus métodos visibles?

2. Diseñar e implementar justo lo suficiente para conseguir que la prueba funcione. Tendrás que diseñar una cantidad suficiente de la implementación para que estas pruebas y todas las anteriores funcionen.

3. Repetir.

4. Si siempre ves la posibilidad de hacer el diseño más simple, hazlo.”[1]

2.4 Desarrollo

El desarrollo conlleva una estrategia que comienza con la planificación de la iteración. La integración continua reduce los conflictos del desarrollo y crea un final natural para un episodio del desarrollo. La propiedad colectiva anima a todo el equipo a mejorar el sistema y finalmente la programación en parejas mantiene unido todo el proceso.

2.4.1 Integración continua

El código no puede permanecer sin integrarse por mucho tiempo. Al final de cada episodio de programación, el código debe ser integrado con la última versión y todas las pruebas deben funcionar al 100%.

No sería posible trabajar con este estilo de integración, si el tiempo para realizar la misma fuera de un par de horas. Es importante tener herramientas que soporten un ciclo rápido de integración/construcción/pruebas. También es necesario tener un conjunto de pruebas que se ejecuten en unos pocos minutos.

La integración continua reduce significativamente los riesgos del proyecto. Si dos personas tienen diferentes ideas sobre la forma o el funcionamiento de una parte del código, se sabrá en horas. No se dará el caso de dedicarle varios días a encontrar un error que fue cometido en algún momento de las últimas semanas.

2.4.2 Propiedad colectiva

Es el concepto de permitir que cualquier persona pueda cambiar cualquier parte del código del sistema en cualquier momento. Esta propiedad del desarrollo se puede mantener si las pruebas bien documentadas y su calidad es muy buena.

Este concepto de propiedad colectiva permite que el código complejo no dure mucho tiempo, esto se debe a que, como todo el mundo puede ver cualquier parte del sistema, este código complejo se encontrará tarde o temprano. Al encontrarlo, alguien intentará simplificarlo. Una vez el nuevo código haya pasado las pruebas, el código complejo será desechado.

Cuando el programador sabe que el código que se está realizando va a ser utilizado por un compañero, se detiene a pensar dos veces antes de agregar un código complejo al sistema.

La propiedad colectiva tiende a esparcir el conocimiento del sistema sobre todo el equipo lo cual reduce los riegos del proyecto.

2.4.3 Programación en parejas

La programación en parejas es un dialogo entre dos personas que intentan simultáneamente programar (y analizar, diseñar y probar) y comprender juntos como programar mejor.

2.4.4 Recodificación

Los programadores reestructurarán el sistema sin cambiar su comportamiento para eliminar la duplicidad, mejorar la comprensión, la sencillez o añadir flexibilidad del código fuente.

El éxito de la recodificación proviene de los siguientes puntos.

- Se debe estar acostumbrado a la propiedad colectiva del código y no se tenga inconvenientes en hacer cambios si son necesarios.

- Tener estándares de codificación, para que no se tenga que cambiar el formato del código antes de hacer la recodificación.

- Codificar en parejas para que se tenga más valentía a la hora de afrontar una mejora difícil, y sea menos probable que se dañe algo sin querer.

- Se debe tener un diseño sencillo para que la recodificación no tenga dificultad.

- Tener pruebas, de esta forma de disminuye la probabilidad de dañar algo sin querer.

- Tener integración continua, ya que si se ha dañado algo, que no está directamente asociado con el diseño que se ha tocado, o la recodificación entra en conflicto con algo que funciona se sepa en cuestión de horas.

2.4.5 Estándares de codificación

Ya que se va a tener a varios programadores cambiando de una parte del sistema a otra distinta, intercambiando compañeros varias veces al día y haciendo recodificación en otras partes del código de forma constante, no es posible tener diferentes tipos de normas de codificación.

El estándar debería exigir la menor cantidad de trabajo posible y ser consistente. Un estándar de codificación muy recomendado y utilizado es de java.

2.5 Pruebas

Las pruebas que se deben escribir en la programación extrema deben ser aisladas y automatizadas. Cada prueba no debe interactuar con las otras pruebas que se escriben, para evitar el problema que una prueba falle y cause otros fallos.

Las pruebas tienen mayor valor cuando el estrés crece, ya que el juicio humano empieza a fallar, al automatizarlas se evita el factor humano, Las pruebas deben devolver una condición de aprobado o rechazado del comportamiento del sistema.

Es muy difícil probar absolutamente todo, por lo que se debe probar cosas que podrían fallar. Si el código es tan sencillo que posiblemente no puede fallar, entonces no se debe escribir una prueba para esto. La experiencia es lo que ayuda al programador a identificar que clase de pruebas tienden a merecer la pena y cuales no.

Las pruebas provienen tanto del programador como del cliente, ellos deben trabajar en conjunto para lograr el éxito de las pruebas.

2.5.1 Los programadores

El programador escribe pruebas método-a-método bajo las siguientes circunstancias.

- Si la interfaz para un método no está clara, debe escribir una prueba antes de escribir el método.

- Si la interfaz está clara, pero imagina que la implementación podría ser un poco menos complicada, debe escribir una prueba antes de escribir el método.

- Si piensa en una circunstancia no usual en la cual el código debería funcionar conforme está escrito, debe escribir una prueba para comunicar la circunstancia.

- Si encuentra un problema posterior, debe escribir una prueba que aísla el problema.

- Si está haciendo recodificación en algún código, y no está seguro de cómo debe ser su comportamiento y todavía no hay una prueba para dicho comportamiento, debe escribir una primera prueba.

Estas, también son conocidas como pruebas de unidad y no se puede decir que algo medio pasó la prueba, la unidad o funciona o no funciona, no hay términos medios.

2.5.2 Los clientes

Los clientes escriben pruebas historia-a-historia. La pregunta que necesitan hacer es "¿Qué tendría que probar antes de estar seguro que esta historia se ha completado?". El cliente establece diferentes escenarios con los cuales se puede probar el producto.

Estas son pruebas funcionales y aunque cada escenario tiene unidades en común otras no lo son, por lo que, la prueba funcional es un éxito o un fracaso, no es discreto. Conforme se vaya cerrando una versión, el cliente necesitará clasificar las pruebas funcionales que fallan. Algunas serán más importantes de corregir que otras. La idea, es llegar al valor más cercano del 100% de aceptabilidad.

El equipo XP debe tener al menos una persona dedicada a hacer pruebas, sin importar el tamaño del equipo. El tiene que ayudar al cliente a escribir las pruebas funcionales y traducir los datos de prueba a prueba. El, también utiliza las pruebas ideadas por el cliente como punto de partida para las variaciones que probablemente harán que falle el *software*.

2.5.3 Otras pruebas

Las pruebas de unidad y funcionales son el centro de la estrategia de pruebas de la programación extrema. El equipo de programación extrema debe reconocer cuando va por más camino y debe utilizar cualquier otro tipo de prueba que pueda ayudar. Algunas que se utilizan se describen a continuación:

2.5.3.1 Prueba paralela

Es una prueba diseñada para probar que el nuevo sistema funciona exactamente como el viejo. La prueba muestra cómo el nuevo sistema difiere del viejo sistema.

2.5.3.2 Prueba de tensión

Diseñada para simular la peor carga posible. Estas son recomendadas para sistemas complejos donde las características de rendimiento no son fáciles de predecir.

2.5.3.3 Prueba de *monkey*

Está diseñada para asegurar que el sistema actúa de forma sensible frente a entradas sin sentido.

2.5.3.4 Pruebas recuperación

Diseñada para forzar el fallo del *software* de muchas formas y verifica que la recuperación se lleva a cabo apropiadamente.

2.6 La gestión

Hay que tener cuidado con la estrategia que se utilice, ya que no es posible que una persona sea la que tome todas las decisiones. No existe una persona que sepa lo suficiente para hacer un buen trabajo y tomar todas las decisiones. Pero tampoco se puede dejar al equipo de programación extrema sin supervisión. Es necesario tener a alguien con una visión superior que pueda influir al equipo cuando éste se desvíe.

Se debe tener un director de proyecto que exponga de una forma clara lo que se necesita hacer y no asignar trabajo. Este director de proyecto debe mantener una relación basada en la sinceridad con los programadores ya que ellos tienen la voluntad de hacer un buen trabajo.

El director debe tener una mentalidad de “estoy consiguiendo ayudar a estos tipos a hacer su trabajo” y no de “estoy intentando conseguir que mi equipo haga un trabajo decente”. El debe proporcionar continuamente indicadores de lo que ha sucedido en cada incremento en vez de dar un manual de políticas al principio.

El director de proyecto necesita tomar la iniciativa en la adaptación de la programación extrema a las condiciones particulares, debe saber cómo la cultura de la programación extrema choca con la cultura de la empresa y encontrar la forma de resolver aquello que no se adapte.

2.6.1 Control

En cualquier proyecto es necesario controlar lo que se está haciendo. Se puede hacer un sin fin de estimaciones, pero si no se mide lo que realmente sucede frente a lo que se estimó no se aprenderá nada.

Se deben reunir lo que las métricas estén controlando en un momento dado y asegurarse que el equipo es conciente de lo que se estaba midiendo realmente sin olvidar lo que había estimado.

El control necesita llevarse a cabo sin muchos costes, la distracción continua del equipo puede llegar a perjudicar el proyecto. Se recomienda que se recojan los datos reales de desarrollo dos veces por semana.

2.6.2 40-horas semanales

Para mantener el ánimo del personal, se debe tener como regla no trabajar más de 40 horas a la semana. Una persona puede permanecer concentrada durante 35 horas, otra durante 45. Pero ninguna puede ser capaz de hacerlo 60 horas semanales durante varias semanas y, todavía sentirse fresco, creativo, prudente y confiado.

Las horas extras son un síntoma de un problema en el proyecto. El problema puede residir en que la estimación de trabajo fue mal hecha, por lo que es necesario reevaluar si la asignación de recursos fue correcta. Si el problema reside en el personal, se tiene que evaluar la razón por la cual no rinde el equipo y hacer lo posible para que no se retrase el proyecto.

2.6.3 El cliente *in-situ*

El cliente debe ser miembro del equipo de desarrollo, y estar disponible para responder a las preguntas, resolver discusiones, y fijar prioridades a pequeña escala. Este cliente debe ser alguien que utilizará realmente el sistema cuando esté en producción Si se está construyendo un sistema de servicios al cliente, éste será un cliente representativo del servicio. Si se está haciendo un sistema de gestión de bonos, el cliente debe ser un agente comercial de bonos.

Hay que ser un buen negociador ya que los usuarios del sistema bajo desarrollo son muy valiosos para cederlos al equipo de desarrollo. Se debe decidir que es más valioso, tener el *software* trabajando pronto y mejor o tener el rendimiento de una o dos personas. Analizando esto, se puede llegar a la conclusión, que si el sistema no aporta más valor al negocio que tener una persona trabajando más tiempo, quizás el sistema no debería ser construido.

2.7 Roles del personal

La programación extrema está basada en el trabajo en equipo y éste equipo está formado por programadores, el cliente, el encargado de pruebas, el controlador, el preparador, el consultor, y el jefe.

2.7.1 El programador

Su principal preocupación es la de ser comunicativo con otras personas, ya que la metodología requiere la programación en parejas y resultaría difícil si no se comunicara con ella.

Debe tener una disposición de aprender como enseñar ya que no hay que ser egoístas con el conocimiento del lenguaje de programación.

El, debe tener presente que, si algún componente de comunicación esencial no se ha hecho, no se ha terminado. Debe tomar como base que las pruebas le van a indicar cuando termina con la codificación. Estas pruebas deben mostrar algún aspecto esencial del *software*.

Debe ser lo más simple posible y que funcione, se puede ayudar afrontando el problema con un enfoque de “divide y vencerás”

El programador debe dejar a un lado el concepto de propiedad individual y aceptar que otra persona cambie o modifique lo que el había escrito.

2.7.2 El cliente

Es un factor muy importante ya que únicamente él conoce qué hay que programar. El cliente deba aprender a escribir buenas historias y a estar cómodo influyendo en un proyecto sin ser capaz de controlarlo.

Debe tener la capacidad de tomar decisiones y conocer de qué forma se podrá dar más valor a su negocio.

El cliente es un factor muy importante para hacer las pruebas ya que él es el encargado de escribir las pruebas funcionales, debe trabajar estrechamente con el equipo para aprender qué clase de cosas son útiles para probar.

2.7.3 El encargado de pruebas

Es responsable de ayudar al cliente a escoger y escribir pruebas funcionales, también debe hacer funcionar las pruebas funcionales regularmente y poner los resultados en un lugar destacado.

No hay que olvidar que el encargado de pruebas no es una persona aislada, dedicada a detectar fallos en el sistema, también debe asegurarse que las herramientas de pruebas funcionen correctamente.

2.7.4 El controlador

Se puede tomar como la conciencia del equipo. Debe hacer muchas estimaciones y observar como la realidad conforma esas suposiciones.

Para verificar, debe observar al conjunto del sistema y dar información al equipo si en algún momento van en el rumbo incorrecto o si deben hacer algo para adaptarse a lo estimado para dicha iteración.

Tiene que recaudar datos para conocer como va funcionado el equipo, los resultados de las pruebas funcionales y de las estimaciones que ya se habían hecho. Todo para poder aprender más del equipo y preparase mejor para futuras iteraciones y/o proyectos.

2.7.5 El preparador

Es el responsable del proceso en conjunto y de entender con mucha mayor profundidad la aplicación, debe identificar que practicas alternativas pueden ayudar al conjunto actual de problemas; como están utilizando otros equipos XP y como relacionan con la situación actual.

Hace su mejor trabajo cuando interviene indirectamente, si ve un error en el diseño, primero evalúa si es lo suficientemente importante como para que intervenga.

Lo primero que se tiene que decidir, es si el problema que se ve de tal tamaño necesite aceptar el riesgo de intervenir. Cada vez que se guía, el equipo se vuelve menos independiente. Demasiada dirección hace que se pierda la habilidad del equipo de trabajar sin el, dando lugar a una baja productividad, una baja calidad y una baja moral.

Debe ser una persona con bastante conocimiento XP para poder detectar cuando está fallando el proceso. El rol del preparador disminuye conforme madura el equipo.

2.7.6 El consultor

El consultor es un especialista, es necesario cuando el equipo necesita un conocimiento técnico profundo en un área.

Tiene como objetivo enseñar al equipo a resolver un problema, debe estar dispuesto a que lo cuestionen ya que las propiedades de la programación extrema hace que se busque la solución más sencilla posible.

2.7.7 El gestor

Es el vínculo entre clientes y programadores, ayuda a que el equipo trabaje efectivamente creando las condiciones adecuadas. Su labor esencial es de coordinación.

Tanto del gestor como el preparador deben ser líderes transformacionales, deben inspirar a los demás con la visión correcta del proyecto, deben promoverla en contra de la resistencia. Las características comunes de los líderes transformacionales son:

- Visionario.
- Inspirador.
- Amable.
- Considerado.
- Digno de confianza.
- Seguro de si.

3 COMO IMPLANTAR LA PROGRAMACIÓN EXTREMA EN UN EQUIPO DE DESARROLLO DE *SOFTWARE*

3.1 ¿Por qué cambiar a programación extrema?

"(La programación extrema)… es un método ligero para que equipos de tamaño mediano a pequeños desarrollen software frente a requisitos imprecisos y muy cambiantes…"[2].

La programación extrema ayuda a llevar un proyecto en el cual los cambios de los requerimientos son una constante, el cliente tiene una muy vaga idea de lo que desea por lo cual se va a ver beneficiado ya que él forma parte del equipo de desarrollo y ayuda a guiar el proyecto para satisfacer sus necesidades que, con el transcurso del tiempo, le van a quedar más claras.

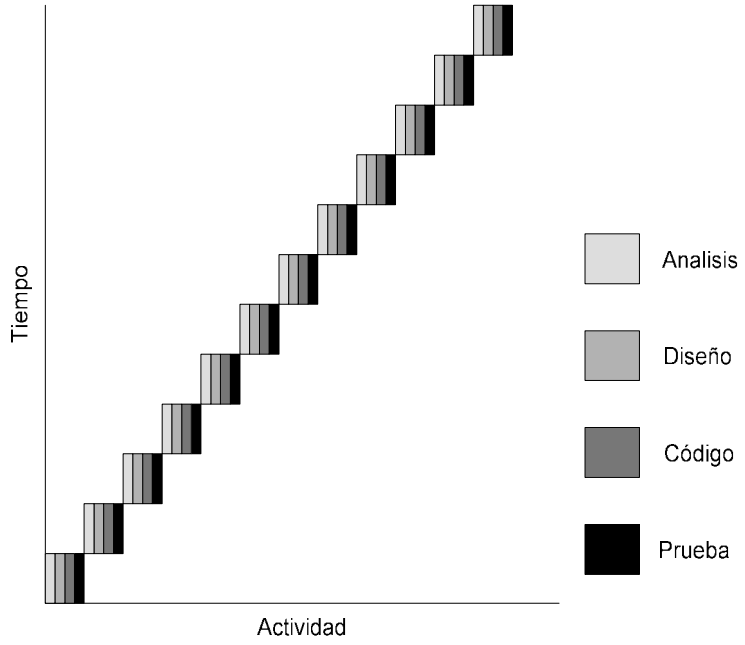
Esta metodología responde a los cambios más que al seguimiento estrictamente un plan. La habilidad de responder a los cambios que puedan surgir a los largo del proyecto (cambios en los requisitos, en la tecnología, en el equipo, por mencionar algunos) determina también el éxito o fracaso del mismo. Por lo tanto, la planificación se hace de una forma flexible y abierta.

A diferencia de otras metodologías, el cliente obtiene un *release* de forma rápida que le proporciona valor a su negocio, no debe espera que el producto este terminado para darse cuenta de otros beneficios que puede obtener del sistema de *software*. Un sistema usable, aunque mínimo, puede entrar en producción pronto. El cliente puede cambiar sus necesidades de acuerdo a los cambios en el negocio, y también aprender cómo se usa el sistema en realidad.

El código que se obtiene es más confiable ya que se este es elaborado por parejas y el evaluado por medio de pruebas constantes las cuales no son elaboradas únicamente por el programador sino también por el cliente. Esto permite una retroalimentación más rápida no solo de los problemas que se obtuvo sino también de otras historias que puedan beneficiar a la empresa.

El desarrollo del sistema sigue el modelo de la siguiente gráfica.

Figura 16. Modelo de desarrollo en la programación extrema

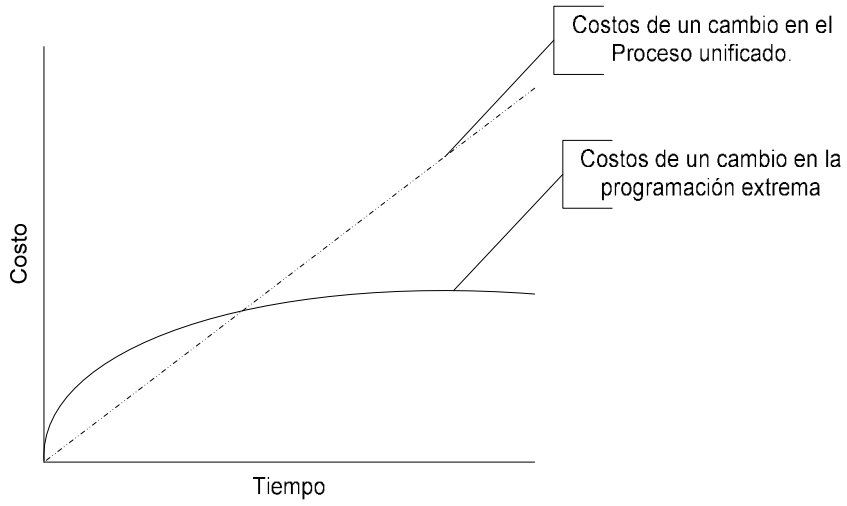


La simplicidad del código generado ayuda a tener un mantenimiento más ordenado, el cual puede ser elaborado por cualquier miembro del equipo ya que se siguen estándares de programación y se utiliza la recodificación.

Las pruebas de integración, que se hacen de forma continua permiten encontrar errores antes de haber avanzado mucho en el proyecto lo cual reduce los riesgos de dar un producto defectuoso al cliente.

Todo esto hace que los costos asociados al cambio se vean reducidos drásticamente a comparación de metodologías tradicionales.

Figura 17. Costos del cambio en un proyecto



Una ventaja del método es de hecho su peso ligero. Se hace únicamente lo que esta programado hacer, la simplicidad es una obligación. Los procesos más simples son más probables de ser seguidos cuando uno no está acostumbrado a ningún proceso en absoluto.

Se debe tener presente que el cliente quiere ver algo funcionando no un montón de documentos que le indiquen lo que se va hacer.

En la siguiente figura se aprecian las algunas características de la programación extrema y otras metodologías tradicionales de desarrollo de *software*

Figura 18. Comparación de metodologías

Programación extrema	Metodologías tradicionales
Basadas en heurística provenientes de prácticas de producción de código	Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo.
Especialmente preparados para cambios durante el proyecto	Cierta resistencia los cambios
Proceso menos controlado con pocos principios	El cliente interactúa con el equipo de desarrollo mediante reuniones
No existe contrato tradicional o al menos es bastante flexible	Grupos grandes y posiblemente distribuidos
Pocos artefactos	más artefactos
Pocos Roles	más roles
Menos énfasis en la arquitectura del *software*	La arquitectura del *software* es esencial y se expresa mediante modelos

3.2 Características deseables en un equipo de desarrollo

La gente es el principal factor de éxito de un proyecto *software*. Es más importante construir un buen equipo que construir el entorno. Muchas veces se comete el error de construir primero el entorno y esperar que el equipo se adapte automáticamente. Es mejor crear el equipo y que éste configure su propio entorno de desarrollo en base a sus necesidades.

El tamaño del equipo puede variar entre 3 a 20 personas, hay que tener en cuenta que a mayor número de personas más difícil va a ser poder establecer la comunicación.

El equipo debe estar conformado de personas carismáticas, que busquen áreas de entendimiento mutuo con los integrantes del equipo. Estas personas deben poder agregar valor al equipo.

Los programadores seniors son aquellos que ya tienen experiencia y por lo regular tienen una buena formación. Este no debe presentar una actitud egoísta frente a sus compañeros, debe ser humilde. Debe tener una actitud que fomente el espíritu del equipo y de la confianza como para poder preguntar algo sin sentirse apenado.

Se puede contar con programadores *juniors*, que tiene un excelente conocimiento del lenguaje o alguna tecnología novedosa, este conocimiento se lo debe pasar al programador *seniors* y este a su vez le brindará el conocimiento para relacionarse en el grupo y como trabajar con la metodología.

Cuando se quiere contratar y retener a gente capaz, hay que reconocer que son profesionales competentes. Deben aceptar la libertad que da la metodología y no abusar de ella.

Al principio se puede sentir mal a estar programando en parejas, pero se debe estar conciente que el no tener el teclado de la computadora no significa que va a perder el tiempo. Se debe ayudar siempre con ideas de diseño, arreglar errores léxicos y semánticos que se den en la programación.

Todos los miembros deben tener una buena habilidad de comunicación, no solo de forma verbal sino también escrita ya que deben dar a conocer sus ideas a su pareja de programación, en las historias y en el código fuente.

La dinámica del liderazgo es muy importante, el gestor y el preparador son los roles que más se visualizan como líderes en el equipo de trabajo, éstos deben tener poder de retribución para poder satisfacer las necesidades de sus seguidores. El consultor tiene un poder experto y al igual que el poder del gestor y el preparador, debe utilizarlo para lograr el bien del proyecto, tratando de obtener una respuesta de compromiso hacia el proyecto y no hacia la persona.

Debido a la libertad de trabajo que propone la programación extrema, es mejor para el proyecto, que los líderes y la administración tengan una opinión de Teoría Y. En resumen se puede decir que la Teoría Y comprende de las siguientes propuestas y opiniones:

- La gerencia es responsable de organizar los elementos de la iniciativa productiva.

- Los empleados no son pasivos por naturaleza ni se resisten a las necesidades organizacionales.

- La motivación, el potencial de desarrollo, la capacidad para asumir responsabilidades y la preparación para orientar las acciones propias hacia los objetivos organizacionales están presentes en el empleado; la gerencia no los pone ahí. Sin embargo, es responsabilidad de ésta hacer posible que la gente reconozca y desarrolle estas características humanas en su persona.

- La tarea esencial de la gerencia consiste en disponer las condiciones organizacionales y los métodos de operación de tal manera que las personas puedan lograr sus propias metas al orientar sus esfuerzos hacia la consecución de los objetivos organizacionales.

La programación extrema se desarrollará de mejor forma si se plantea un liderazgo de participación o un liderazgo delegativo según el modelo de liderazgo situacional de Hersey y Blanchard,

3.3 Como migrar a programación extrema

Antes de hacer el cambio de debe establecer si esta metodología es la correcta para el proyecto que se desea trabajar. Esta metodología responde más a los cambios y se utiliza con equipos de desarrollo pequeños y medianos.

El ingeniero en sistemas sabe la resistencia al cambio que sufre por parte del cliente y debe estar conciente que él también la sufre. Debe tener valentía para modificar a su equipo a trabajar con una metodología que tiene estos cuatro valores:

- Comunicación.
- Sencillez.
- Realimentación.
- Valentía.

y las siguientes actividades básicas

- Codificar.
- hacer pruebas.
- Escuchar.
- Diseñar.

Esto en conjunto ayuda a cumplir con los principios básicos que son

- Realimentación rápida.
- Asumir la sencillez.
- Cambio incremental.
- Aceptar el cambio.
- Trabajar con calidad.

Este tipo de proceso adaptable requiere un tipo diferente de relación con el cliente. La forma de hacer el contrato debe cambiar ya que normalmente se contrata a una empresa para hacer el desarrollo del *software* con un contrato de precio fijo. Se le dice a los desarrolladores lo que se quiere, se negocia, se acepta una oferta, y entonces la carga queda en la empresa de desarrollo para construir el *software*.

Un contrato a precio fijo requiere requisitos estables y por tanto procesos predictivos. Los procesos adaptables y los requisitos inestables implican que no se puede trabajar con la noción usual de precio fijo. Tratar de encajar un modelo de precio fijo a un proceso adaptable acaba con una experiencia insatisfactoria. El equipo XP debe ofrecer algo más parecido a una suscripción a un servicio. El equipo trabajará a velocidad tope para el cliente durante una cierta cantidad de tiempo. Al comienzo de cada iteración el cliente tiene una posibilidad formal para cambiar la dirección e introducir historias completamente nuevas que ayuden a su negocio.

No solo se tiene que introducir al cliente a esta nueva forma de contrato sino que también debe tomar una participación más activa en el proceso de desarrollo de *software*. Debe estar dispuesto a ayudar al equipo de programación extrema y saber que ellos lo van a ayudar creando un producto de mayor calidad gracias a las pruebas que diseñaron en conjunto.

El entorno físico debe ser un ambiente que permita la comunicación y colaboración entre todos los miembros del equipo durante todo el tiempo, cualquier resistencia del cliente o del equipo de desarrollo hacia las prácticas y principios puede llevar al proceso al fracaso.

El equipo debe dejar el pensamiento individualista a un lado y saber que el éxito o fracaso del proyecto se debe a el equipo no a alguien específico.

Las pruebas de funcionabilidad como de integración son importantes para el proceso y para que no quiten mucho tiempo se debe utilizar una herramienta que ayude en esto. Debido a que la mayoría de equipos extremos utilizan java, se ha popularizado el uso de Junit.

CONCLUSIONES

1. Las metodologías imponen un proceso disciplinado sobre el desarrollo de *software* con el fin de hacerlo más predecible y eficiente. Lo hacen desarrollando un proceso detallado con un fuerte énfasis en planificar inspirado por otras disciplinas de la ingeniería.

2. No existe una metodología universal para hacer frente con éxito a cualquier proyecto de desarrollo de *software*. Toda metodología debe ser adaptada al contexto del proyecto (recursos técnicos y humanos, tiempo de desarrollo, tipo de sistema, etc).

3. Una de las limitaciones de esta metodología es cómo maneja equipos grandes. Como muchas nuevas tendencias, ellos tienden a ser usados primero a escala pequeña antes que a gran escala. También a menudo se han creado con énfasis en equipos pequeños. La XP explícitamente dice que está diseñada para equipos de no más de veinte personas.

4. La lluvia de ideas que se genera con la programación en parejas crea diseños de mayor calidad en menor tiempo.

5. Un equipo de *software* puede reducirse en tamaño sin reducir su productividad total.

6. La motivación de los miembros del equipo aumenta al trabajar en conjunto para resolver un problema y lograrlo.

7. La seguridad del programador aumenta al saber que el trabajo ya fue revisado por su compañero de programación.

8. Las prácticas de la programación extrema que son compatibles con el proceso unificado son:

   - La forma en que se realiza la planificación de la programación extrema.
   - Diseñar las pruebas primero y recodificación.
   - La integración continúa.
   - El cliente in situ.
   - Los estándares de codificación.
   - Cuarenta horas semanales de trabajo.
   - Programación en parejas.

9. Las prácticas de la programación extrema que no son compatibles con el proceso unificado son:

   - Propiedad colectiva del código fuente.
   - Recodificación en sistemas grandes.
   - Entrega de Pequeñas versiones.

10. El sistema de educación está adaptado para el éxito individual, los sistemas de recompensa en muchas compañías con evaluaciones y subidas individuales también fomentan el pensamiento individual.

11. Se debe derribar muros emocionales, para que la programación extrema tenga éxito, hay que ser comunicativo y sincero de esta manera se obtiene el respeto del equipo.

12. Debe existir una interacción constante entre el cliente y el equipo de desarrollo. Esta colaboración entre ambos será la que marque la marcha del proyecto y asegure su éxito.

13. La programación extrema, además de ofrecer sistemas de calidad, plantea nuevas alternativas de respuesta a la solución de problemas complejos y, en la mayoría de los casos, facilita las modificaciones en las aplicaciones para lograr su extensibilidad.

14. Los programadores son concientes de la importancia de documentar el código fuente y respetar los estándares de codificación pero la urgencia que viene de una mala estimación hace que dichas actividades se desechen para entregar el hito en tiempo. Esta urgencia también hace que los equipos de desarrollo no utilicen a completo las herramientas CASE.

15. Es frecuente que un cliente observe la verdadera oportunidad de realizar un sistema útil después que éste se le ha entregado. La programación extrema permite tener una temprana retroalimentación del cliente para poder realizar cambios mientras se tiene tiempo y mejorar la aceptación del cliente.

16. La metodología XP fue creada en respuesta del problema de cambio de requerimientos. Es muy frecuente que el cliente no tenga una clara idea de lo que el sistema debe hacer. Se tiene un sistema cuya funcionabilidad cambia cada pocos meses. En muchos sistemas la única constante es el cambio dinámico de los requerimientos.

RECOMENDACIONES

1. Este proyecto final de carrera da a conocer tres modelos de desarrollo de *software*, ciclo de vida clásico, proceso unificado y programación extrema, sin embargo existen otras metodologías de *software* que pueden ayudar a un encargado de proyecto a tener las bases para salir adelante. Es recomendado que se realice un proyecto final de carrera que pueda mencionar otras metodologías para que tenga mayor conocimiento de ellas. Algunas de estas metodologías son:

   - Metodología cristal.
   - Código abierto.
   - El Desarrollo de *software adaptable* de Highsmith.
   - Scrum.
   - Desarrollo manejado por rasgos.
   - DSDM (Método de desarrollo de sistema dinámico).

2. La mayoría de personas, incluyendo a aquellas que tiene conocimientos informáticos, tienen un modelo mental en el que el empleado del área de informática debe estar aislado en su cubículo haciendo algo que únicamente él entiende. Este aislamiento es algo que se debe evitar ya que el éxito del departamento de informática en cualquier organización está basado en la comunicación con los otros departamentos. El cambio de este modelo mental debe empezar con el estudiante de ingeniería en sistemas, se le debe enseñar a trabajar en un grupo lo más pronto posible ya que vivimos en sociedad donde raras veces una persona sin ayuda de los demás tiene éxito.

REFERENCIAS

1. Beck, K. 2002 **Una explicación de la Programación Extrema. Aceptar el cambio**. Addison-Wesley, 2002. p.106

2. Beck, K. 2002 **Una explicación de la Programación Extrema. Aceptar el cambio**. Addison-Wesley, 2002. p. XV

BIBLIOGRAFÍA

BECK, Kent. **Una explicación de la programación extrema. Aceptar el cambio** España 2002, Ed. Pearson Educación S.A. . ISBN 84-7829-055-9.

BOOCH, Grady. **Analisis y diseño orientado a abjetos con aplicaciones** 2ª ed. E.E.U.U 1996, Ed. Addison-Wesley Iberoamericana, S.A. ISBN 0-201-60122-2.

Clarkware, http://www.clarkware.com/software/JDepend.html octubre 2003

Economist, http://www.economist.com/displaystory.cfm?story_id=779459 agosto 2003

Extreme programming, http://www.extremeprogramming.org/ febrero de 2003

fawcette technical publications, http://www.fawcette.com/interviews/beck_cooper/page1.asp septiembre 2003

HELLRIEGEL, JACKSON Y SLOCUM. **Administración, un enfoque basado en competencias**. 9ª. ed. México: International Thomson Editores, S.A. ISBN 970-686-197-1.

Iternum, http://www.iternum.com/developer/journal/opinions/xtreme.jsp diciembre 2003

Object Mentor, http://www.objectmentor.com/writeUps/TestDrivenDevelopment octubre 2003

O'Reilly Network, http://linux.oreillynet.com/pub/a/linux/2001/05/04/xp_intro.html junio 2003

PRESSMAN, Roger **Ingeniería de Software** 4ª ed. España 1998, Ed McGraw-Hill/ interamericana ISBN 84-481-1186-9.

Rational, http://www.rational.com/ junio 2003

ROJAS, Claudia, Curso de Análisis y Diseño 1, notas del Curso 2002 U.S.A.C. Guatemala.

SENN, James. **Análisis y diseño de sistemas de información**. 2a ed. México: Ed. McGraw Hill 2002. ISBN 968-422-991-7.

Software reality, http://www.softwarereality.com/lifecycle/xp/conclusion.jsp agosto 2003

SUMMERVILLE, Ian. **Ingeniería de software** 6ª ed. Mexico 2002. Ed.Person Educación ISBN 970-26-0206-8.

The Portland Pattern Repository, http://c2.com/cgi/wiki?XpGlossary febrero de 2003

VILAS, Ana Fernandez , http://www-gris.det.uvigo.es/~avilas/UML/node60.html noviembre 2003

Xprograming, http://www.xprogramming.com/ febrero de 2003

[1] Beck, K. 2002 Una explicación de la Programación Extrema. Aceptar el cambio. Addison-Wesley, 2002.

[2] Beck, K. 2002 Una explicación de la Programación Extrema. Aceptar el cambio. Addison-Wesley, 2002