C.F.G.S. DESARROLLO DE APLICACIONES WEB

Módulo Profesional
ENTORNOS DE DESARROLLO

UD 1. DESARROLLO DE SOFTWARE.

Curso : 2018/19

Profesor : Jorge Martín Cabello

Contenido

1. INTRODUCCIÓN

La labor de desarrollo de software es realizada por un programador, o un equipo de programación. El desarrollador de software puede contribuir en la visión general del proyecto más a nivel de aplicación que a nivel de componentes, así como a las tareas de programación individuales. También puede realizar la función de analista o de programador, arquitecto de la aplicación, etc.

El concepto de desarrollo de software incluye:

- *Trabajo en equipo*: los proyectos en la mayor parte de los casos parte de la colaboración entre diferentes desarrolladores, y también de otros tipos de colaboradores, comerciales (definen junto al cliente la finalidad y necesidades del producto), diseñadores gráficos (definen el aspecto de la interfaz gráfica, *GUI*, de la aplicación). En muchos casos los roles de el equipo pueden ser llevados a cabo por desarrolladores.
- *Concepción o diseño a partir de unas condiciones*, especificaciones o requisitos.
- *Pruebas*: investigaciones y técnicas cuyo objetivo es proporcionar información objetiva e independiente sobre la calidad del producto, siendo una actividad más en el control de calidad.
- *Mantenimiento*: abarca la corrección de errores o modificaciones después de que haya comenzado el uso comercial del programa informático.

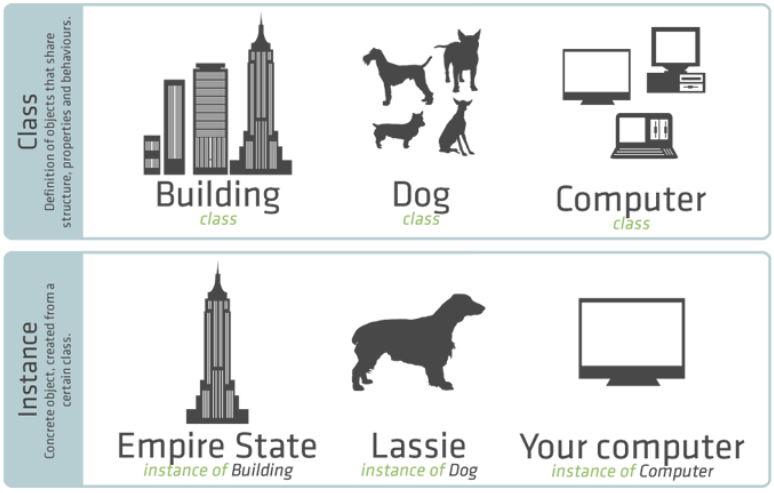
Aspectos sobre el desarrollo de software

Si tuviéramos que plantear una hipotética lista sobre algunas de las necesidades “prácticas” de dominio para un desarrollador, aparte de la formación teórica como matemáticas, lógica, estructuras de datos, ingeniería de software, estructura y arquitectura de computadores , indicaríamos:

- **Programación orientada a objetos**

Aunque haya personas que digan que ya ha alcanzado su límite y empieza a ser desplazada por otros paradigmas, a día de hoy sigue siendo la forma de desarrollar la mayoría de las aplicaciones. El lenguaje no es tan importante (C++, C#, Java), como tener claros y localizados los conceptos: lo que es una clase, un objeto, una clase abstracta, una interface (no tiene nada que ver con interfaz gráfica) , una referencia, un método estático, un método de instancia, la herencia, el polimorfismo, etc.

Además hay un concepto que es especialmente útil conocer: **los patrones de diseño** . Quizás no tanto todos los diferentes patrones en sí, ya que muchas veces están sobrevalorados, pero si el concepto, que nos ayudará a tener un lenguaje común de comunicación con otros desarrolladores. Además la forma de implementar estos patrones ayuda a aplicar el diseño orientado a objetos.



No se trata de trabajar con las últimas tecnologías, que podrían quedarse obsoletas en poco tiempo, sino de comprender perfectamente conceptos necesarios para comprender el funcionamiento de esas tecnologías actuales o futuras.

- **Bases de Datos**

Casi todas las aplicaciones necesitan almacenar información, y en la mayoría de los casos esta termina en una base de datos. Hasta hace poco decir base de datos era sinónimo de hablar de *base de datos relacional*, pero cada vez hay más alternativas, como las bbdd documentales, o las orientadas a objetos.

No es posible mientras nos formamos dominarlas todas, pero igual que algunos conceptos del punto anterior, conviene que las conozcamos y tengamos una idea aproximada de para qué se usan.

Por otra parte sigue siendo muy importante tener una buena base de código SQL y diseño de bases de datos, para lanzar consultas básicas. Además SQL nos aporta una visión de la lógica de conjuntos (consultas de unión) muy importante en la informática.

- **Hardware y redes**

Aunque un desarrollador podría dedicarse únicamente a programar sin saber nada del hardware que ejecuta, la situación real no es así. No es necesario ser capaz de diseñar un microprocesador, pero sí entender que hay diferencias en rendimiento cuando un programa accede a la memoria RAM, a la caché, o a un dispositivo de almacenamiento. Y

más importante aun que no es lo mismo acceder a los datos de nuestro disco local que a un servidor en la otra punta del mundo.

De la misma forma conocer el funcionamiento de las redes de comunicaciones en mayor o menor medida, sobre todo a nivel de protocolos, es muy importante en el desarrollo de aplicaciones que deben funcionar en el mundo real. La diferencia entre un protocolo UDP o TCP, o uno de un nivel más alto como HTTP, las diferencias entre las redes wifi, cableadas o una red de datos móviles, son cuestiones importantes a tener en cuenta. No es necesario saber como se realiza el *handshake* TCP, pero debería ser conocimiento de cualquier informático saber que TCP comprueba si se han recibido los paquetes.

- **Sistemas de control de versiones**

Al igual que ocurre con los lenguajes de programación, no se trata de con qué herramienta o sistema te familiarices, sino de saber usar uno, y entender su función, ya que es el principal método para trabajar simultáneamente en un proyecto entre los componentes de un equipo de desarrollo, poner en común los avances de cada uno, almacenar copias de seguridad de los estados del proyecto, etc.

Git es una de las herramientas básicas junto al manejo de un lenguaje o de un IDE de desarrollo, y todo informático debe conocer cómo subir y bajar código, obtener cambios de otros desarrolladores, crear ramas, etc.

Si aprendemos a usarlo evitaremos tener que guardar nuestros progresos en un pen-drive, dudando de cuál era la última versión que habíamos creado.

- **Programación funcional**

Es un paradigma de programación (al igual que la programación orientada a objetos) que se usa cada vez más, incluso lenguajes tradicionales orientados a objetos como Java o C++, que evolucionan más lento empiezan a implementar características de la programación funcional.

Lenguajes más modernos como Ruby, Python o Javascript ya se han adaptado a este paradigma. Conocer un paradigma que está ganando popularidad a día de hoy, aporta nuevas formas de razonamiento sobre los problemas a resolver en programación.

2. LENGUAJES DE PROGRAMACIÓN

Los podemos clasificar los lenguajes de programación atendiendo a distintos criterios:

2.1. Según su nivel de abstracción

Se entiende como la separación de un elemento de su contexto, o del resto de elementos que lo acompañan. La abstracción se centra en el *¿Qué hace?* más que en el *¿Cómo lo hace?*.

Lenguajes de Alto Nivel

Los lenguajes de alto nivel son aquellos que expresan sus algoritmos de forma comprensible por la capacidad humana, en lugar de la capacidad de las máquinas. Cuando hay un gran nivel de abstracción tratamos con variables, matrices, objetos, aritmética, funciones, en lugar de con registros del procesador, direcciones de memoria, o las pilas de llamadas. Generan un código más sencillo y comprensible, permitiendo que en algunos casos pueda ser para diferentes máquinas o sistemas operativos. Permiten utilizar diferentes paradigmas de programación. Ejemplos: Java, Ada, Visual BASIC, VB .NET, C#, FORTRAN, Lisp, Pascal, Python, Perl, PHP, Ruby, SQL

Lenguajes de Nivel Medio

Tienen características que los acercan a los lenguajes de bajo nivel, pero a la vez también tiene características de los lenguajes de alto nivel. Se suelen utilizar para la aplicaciónes como la creación de sistemas operativos (p.e. Unix en C) Ejemplos: C, C++

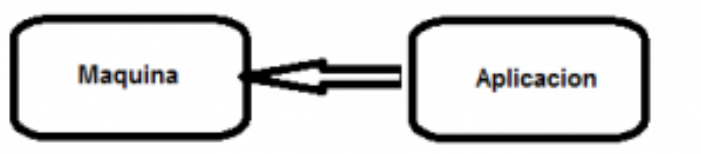
Lenguajes de Bajo Nivel

El bajo nivel de abstracción se centra en lenguajes próximos a los que entiende la máquina, *código máquina*. Ejercen un control directo sobre el hardware (memoria, cpu) y dependen de la estructura física de este. Es por esto que los programas escritos en lenguajes de bajo nivel no son portables entre distintos equipos. Apuntemos que el código

máquina es el código nativo de cada procesador, está organizado en instrucciones y se ejecuta directamente. No es un código escrito por programadores. Ejemplos: código máquina, lenguaje ensamblador.

2.2. Según su forma de ejecución

Lenguajes Compilados



Son aquellos que generan un código resultante que posteriormente será ejecutado. Esto puede darse, creando un archivo ejecutable con el código máquina, preparado para su ejecución directa por la máquina física. Pero algunos lenguajes actuales también permiten compilar su código fuente dando como resultado un código intermedio, almacenado en un fichero. Este fichero es interpretado posteriormente y ejecutado directamente paso a paso (convertido paso a paso a código máquina). Los programas compilados a código nativo en tiempo de compilación, suelen ser más rápidos que los traducidos en tiempo de ejecución. Generan el problema de que el código máquina creado es dependiente de la arquitectura de la plataforma en la que se compilan y para la que se ejecutan. Ejemplos: C, C++, Visual Basic, Fortran, Pascal

Lenguajes Interpretados

Son lenguajes cuyas instrucciones se traducen para ser **ejecutadas por la máquina hardware en el mismo momento de la ejecución**, sin crear ningún código intermedio, ni guardar el resultado de dicha traducción. Son más lentos que los lenguajes compilados debido a la necesidad de traducir a código máquina el programa, instrucción a instrucción, mientras se ejecuta. Debido a esta ejecución en tiempo real, no se traduce la totalidad del conjunto de instrucciones, sino se va traduciendo a medida que se van ejecutando cada una de ellas. Permiten ofrecer al programa interpretado un entorno **no** dependiente de la máquina donde se ejecuta el interprete, sino del propio interprete (máquina virtual). Ejemplos: Html, Php, Python, Ruby, Javascript

Lenguaje Intermedio

Conocemos bajo este concepto el producto de la compilación de algunos lenguajes de alto nivel en un tipo de lenguaje (bytecode). Para mejorar el procesos de optimización o facilitar

la *portabilidad*, algunas implementaciones de lenguajes de programación pueden compilar el código fuente original en una más compacta forma intermedia y después traducir eso al código de máquina. Esto ocurre con lenguajes como Java o C#.

2.3. Según su sistema de tipos

En una clasificación menos convencional y poco consensuada, pero también es una característica importante de los lenguajes de programación. Los tipos de datos de cada lenguaje, representan la naturaleza de los datos que un programa puede emplear. Por ejemplo: valores numéricos (números enteros, decimales, naturales, etc), valores lógicos (verdadero o falso), valores de texto (caracteres, cadenas fijas, cadenas variables), expresiones matemáticas, etc.

- Lenguajes fuertemente tipados (tipado estático): Un lenguaje de programación es fuertemente tipado si no se permiten violaciones de los tipos de datos, es decir, una vez definido el tipo de una variable, no se puede usar como si fuera de otro tipo distinto a menos que se haga una conversión. Cada tipo de datos tiene un *dominio de valores* (rango, valores permitidos) y su tipo de datos controla que no podamos salirnos de su dominio. Para convertir de unos tipos a otros debemos usar sentencias de conversión. Ejemplos: Java, C++, C, C#

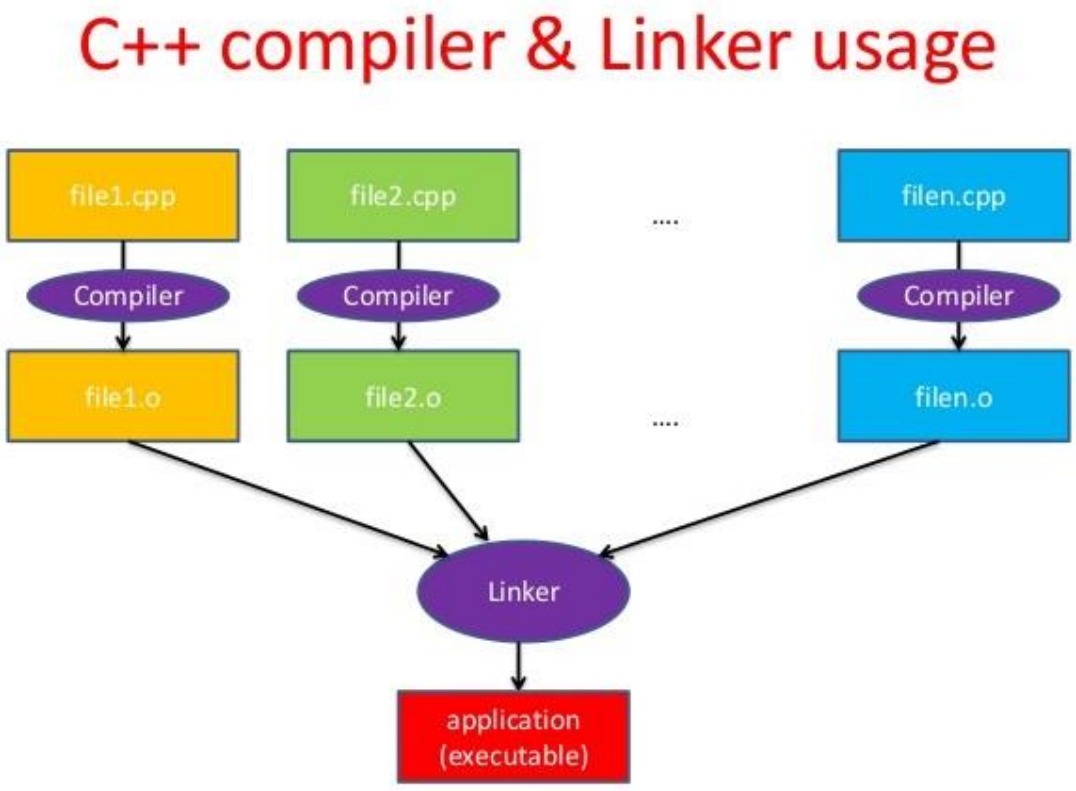
- Lenguajes débilmente tipados (tipado dinámico): Los lenguajes de programación no tipados o débilmente tipados no controlan los tipos de las variables que declaran, de este modo, es posible usar valores de cualquier tipo en una misma variable. Por ejemplo, una función puede recibir indiferentemente como parámetro un valor entero, cadena de caracteres, número flotante, etc. Un lenguaje que no está tipado se refiere a que no está fuertemente tipado. Son lenguajes en los que no se define el tipo de una variable y puede almacenar distintos tipos de datos: números, caracteres, etc. Ejemplo: Javascript, Python

3. CONSTRUCCION DE UN PROGRAMA INFORMÁTICO

Para ver nuestro programa en ejecución, debemos aplicar unos procesos intermedios mediante los cuales nuestro programa cambia de un estado a otro hasta que obtenemos una versión ejecutable. Este concepto se conoce como *Build*.

- **Código Fuente**: Es el código escrito por los programadores utilizando algún editor de texto o algun entorno de desarrollo. Este código no es directamente ejecutable por el ordenador.
- **Código Objeto**: Es el código resultante de compilar el código fuente. No es directamente ejecutable por el ordenador, pero tampoco es entendido por el ser humano. Es un código o representación intermedia de bajo nivel.
- **Código Ejecutable**: Es el resultado de enlazar el código objeto con una serie de rutinas y librerías, obteniendo el código que es directamente ejecutable por la máquina.

3.1. Proceso de compilación



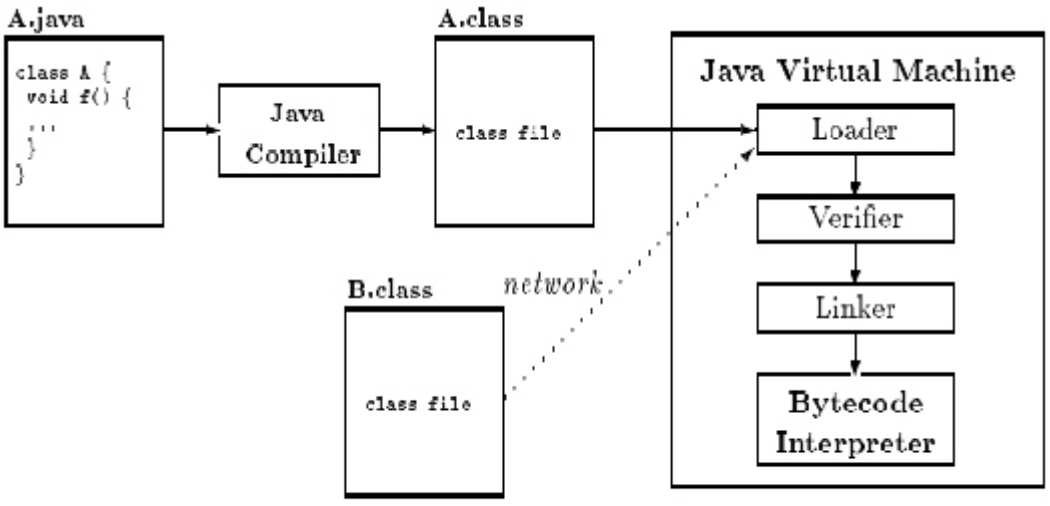
La obtención de un programa ejecutable se lleva a cabo mediante dos programas: el compilador y el enlazador (linker). Aunque difiere dependiendo de la tecnología o plataforma de programación que usemos.

Aun así, la compilación se compone internamente de varias etapas:

- **Análisis léxico**: se lee secuencialmente todo el código fuente agrupándolo en unidades significativas de caracteres (tokens). Son secuencias de caracteres que tienen significado (int, =, void, …). Todos los espacios en blanco, lineas en blanco, comentarios, etc, se eliminan del programa fuente.
- **Analizador sintáctico**: recibe el código fuente en forma de tokens y realiza el análisis sintáctico que determina la estructura del programa; comprueba si las construcciones cumplen las reglas de sintaxis definidas en el lenguaje de programación correspondiente. El proceso es semejante al análisis gramatical de una frase en lenguaje natural.
- **Analizador semántico**: se comprueban que las declaraciones son correctas, se verifican los tipos de todas las expresiones, si las operaciones se pueden realizar sobre esos tipos, si los arrays tienen el tamaño adecuado, etc.
- **Generación del código intermedio**: después del análisis se genera una representación intermedia similar al código máquina con el fin de facilitar la tarea de traducir al **código objeto**.

En la fase de enlazado (linker) se unen todos los códigos objetos resultados de la compilación de todos los ficheros fuentes que forman parte de un programa. También se une el código de los métodos de las librerías usadas. Todos estos códigos objetos se traducen a código máquina creando un programa ejecutable.

3.2. Máquina virtual

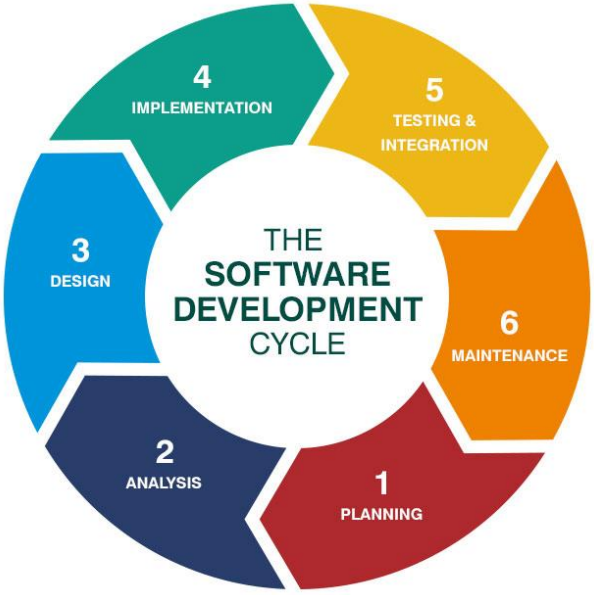


En el caso de que nuestra plataforma use una máquina virtual para ejecutar nuestro código compilado, la máquina se inicia automáticamente cuando se lanza el proceso que se desea ejecutar y se detiene cuando se finaliza. Este tipo de máquinas virtuales se conocen como máquinas virtuales de proceso, y son en sí un proceso normal más dentro del sistema operativo. Su objetivo es el de proporcionar un entorno de ejecución independiente de la

plataforma de hardware y del sistema operativo. El ejemplo más conocido es la *Java Virtual Machine*. La máquina virtual de java se encarga de cargar, verificar, enlazar e inicializar los ficheros *.class* obtenidos del proceso de compilación de los fuentes *.java*.

- En el proceso de carga se buscan los ficheros binarios de los tipos de las clases atendiendo a los identificadores y se crea la clase a partir de esos ficheros binarios.
- En la fase de verificación se comprueban que los bytecodes recibidos no contienen instrucciones que son obviamente dañinas para el sistema.
- Fase de enlace, es el momento en el que se cogen las clases o interfaces y se combinan en un estado de ejecución de la máquina virtual para ser ejecutada.

4. CICLO DE VIDA DEL SOFTWARE



El Desarrollo de Software es una área cuyo comienzo se dio exactamente en el año 1948, año en el que la primera máquina que almacenaba programas pudo ejecutar exitosamente el primero , por lo que es una disciplina que es apenas una recién nacida frente a otras con una base de conocimiento sólida y estable. Durante años el tiempo de vida de un producto acabado (software) era muy largo; durante este tiempo, generaba beneficios a las empresas, para las que era más rentable este producto que las posibles novedades.

A partir de los 80, esta situación empieza a cambiar. La vida de los productos es cada vez más corta y una vez en el mercado, son novedad apenas unos meses, quedando fuera de él enseguida. Esto obliga a cambiar la filosofía de las empresas, que se deben adaptar a este cambio constante y basar su sistema de producción en la capacidad de ofrecer novedades de forma permanente.

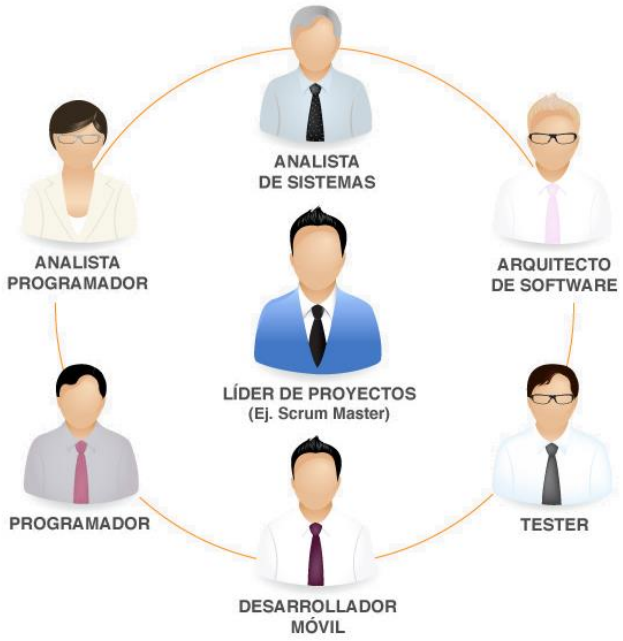
4.1. Fases de desarrollo de un proyecto

Dependiendo de los modelos de desarrollo o de ciclos de vida empleados, hay una serie de actividades que son comunes en el desarrollo de un producto software:

- ***Planificación y diseño***: análisis de requisitos, a través de documentación, de entrevistas con el cliente, o de otros medios. Después de obtener los requisitos del clientes se debe realizar un análisis dentro de contexto del desarrollo. Tratamos de crear un plan sobre qué vamos a hacer y cómo lo vamos a hacer.
- ***Implementación, pruebas y documentación***: La implementación es la construcción de la solución, la programación. La fase de pruebas debe comprobar que toda la implementación cumple con su cometido, incluso puede dirigir la implementación. Implementación y pruebas son fases distintas pero pueden realizar al mismo tiempo, o una antes que la otra dependiendo del modelo de desarrollo seguido. La documentación también es un proceso muy ligado al diseño de las clases y métodos del proyecto, y se pueden realizar al mismo tiempo. Facilita la mejora del código y su mantenimiento a largo plazo.
- ***Despliegue y mantenimiento***: El despliegue se da una vez que tenemos un producto que cumple los requisitos y se implanta en el entorno de producción, en el entorno en el que será usado. Es necesario instruir a los usuarios del producto ya que por naturaleza son reacios al cambio. Por otra parte el mantenimiento del

software se lleva a cabo para resolver problemas que se localizan en el programa o ampliar funcionalidades.

4.2. Los roles dentro de un equipo de desarrollo



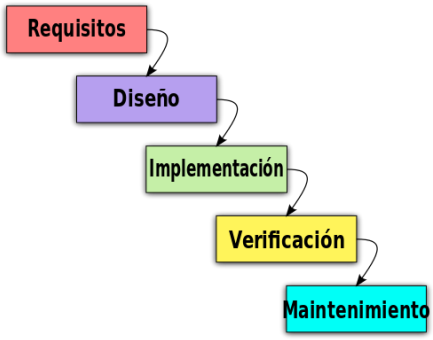
Saber distinguir las obligaciones y limitaciones de cada uno de los roles del equipo ayuda a que el trabajo lo realicen las personas mejor capacitadas para ello, lo que se traduce en mayor calidad. Roles distintos no necesariamente significa personas distintas, sobre todo en equipos muy reducidos. Una persona puede adoptar más de un rol, puede ir adoptando distintos roles con el paso del tiempo, o rotar de rol a lo largo del día. Hagamos un repaso a los papeles más comunes en un proyecto software.

- **Dueño del producto**: su misión es pedir lo que necesita (no el cómo sino el qué), y aceptar o pedir correcciones sobre lo que se entrega.
- **Cliente**: Coincide con el dueño del producto en muchos casos, es el usuario final del producto.
- **Analista de negocio:** También es el dueño del producto porque trabaja codo a codo con el cliente y traduce los requisitos en tests de aceptación para que los desarrolladores los entiendan, es decir, les explica qué hay que hacer y resuelve sus dudas.
- **Desarrolladores:** Toman la información del analista de negocio y deciden cómo lo van a resolver además de implementar la solución. Aparte de escribir código, los desarrolladores deben tener conocimientos avanzados sobre usabilidad y diseño de interfaces de usuario.
- **Administradores de sistemas**: Se encargan de velar por los servidores y servicios que necesitan los desarrolladores.

5. MODELOS DE DESARROLLO

5.1. Modelo de desarrollo en Cascada

Es el más básico de todos los modelos y ha servido como bloque de construcción para los demás paradigmas de ciclo de vida. Está basado en el ciclo convencional de una ingeniería (Se comienza por el principio) y su visión es muy simple: **el desarrollo de software se debe realizar siguiendo una secuencia de fases**.



Cada etapa tiene un conjunto de metas bien definidas y las actividades dentro de cada una contribuyen a la satisfacción de metas de esa fase. Es un modelo de ciclo de vida usado en el software de hace varias décadas, aunque sigue siendo un modelo a seguir ante cierto tipo de desarrollos.

El modelo en cascada abarca las siguientes etapas:

1. ***Análisis***: Ya que el software es siempre parte de un sistema mayor, la fase de análisis comienza con el **Análisis de los requisitos Sistema**, donde se establecen los requisitos de todos los elementos del sistema. Uno de estos elementos es los requisitos de software, que nos lleva al **Análisis de los requisitos de software**. Es el proceso de recopilación de los requisitos que se centra e intensifica especialmente en el software.
2. ***Diseño***: En esta etapa se traducen los requisitos en una representación de software. El diseño se puede basar en dos tipos: **diseño estructurado** y **diseño orientado a objetos**.
   - El diseño estructurado se compone de 4 partes: La **estructura de los datos**, donde se definen las estructuras de datos que se usarán para implementar el software. **La arquitectura del software**, que se centra en la representación de la estructura del software, sus propiedades e interacciones. El **diseño procedimental** se encarga de traducir los elementos estructurales de la arquitectura del software con suficiente nivel de detalle como para que sirva de guía en la generación de código. Por último, **el diseño de la interfaz** describe cómo se comunica el software consigo mismo, con las partes de las que está compuesto, y con los usuarios.
   - El diseño orientado a objetos se centra en el diseño del subsistema, el diseño de las clases y objetos y su jerarquía, la colaboración entre los objetos y las responsabilidades que hay entre clases. Es muy común realizar este diseño empleando el lenguaje de modelado **UML** (*Unified Modeling Language*), el cual está basado en diagramas.

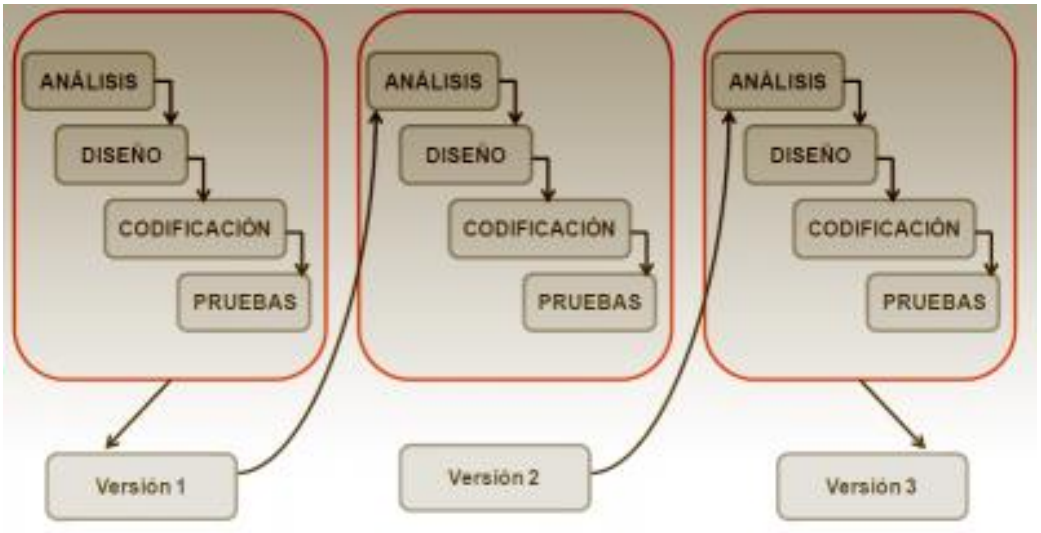
3. ***Codificación o implementación***: el diseño debe traducirse en una forma legible para la máquina. Si el diseño se realiza de una manera detallada, la codificación puede realizarse mecánicamente.
4. ***Pruebas o verificación***: una vez que se ha generado el código comienza la prueba del programa. La prueba se centra en la lógica interna del software y en las funciones externas, realizando pruebas que aseguren que la entrada definida produce los resultados que realmente se requieren.
5. ***Mantenimiento*****:** el software sufrirá cambios después de que se entrega al cliente. Los cambios ocurrirán debidos a que se haya encontrado errores, a que el software deba adaptarse a cambios del entorno externo (sistema operativo o dispositivos periféricos) o a que el cliente requiera ampliaciones funcionales o del rendimiento.

En el modelo vemos una ventaja evidente y radica en su sencillez, ya que sigue los pasos intuitivos necesarios a la hora de desarrollar el software. Pero el modelo se aplica en un contexto, así que debemos atender también a él y saber que:

- Los proyectos reales raramente siguen el flujo secuencial que propone el modelo. Siempre hay iteraciones y se crean problemas en la aplicación del paradigma.
- Es difícil para el cliente establecer todos los requisitos antes de comenzar.
- El cliente no puede ver una versión operativa del proyecto hasta las etapas finales. **Un error no detectado hasta que el programa esté funcionando puede ser desastroso**.

5.2. Modelo de desarrollo Iterativo e Incremental

Se basan en realizar una iteración de varios ciclos de vida en cascada realimentados. En cada iteración se entrega una pequeña parte del software funcional llamada “incrementos”. En general cada incremento se crea sobre aquel que ya ha sido entregado.



En la imagen se observa de forma iterativa distintas fases de desarrollo en cascada para la obtención de un nuevo incremento (versión del programa).

Como ejemplo de software desarrollado bajo este modelo se puede considerar un procesador de textos. En el primer incremento se desarrollan funciones básicas de gestión de archivos y de producción de documentos; en el segundo incremento se desarrollan funciones gramaticales y de corrección ortográfica, en el tercer incremento se desarrollan funciones de paginación, etc.

Ventajas:

- No se necesitan conocer todos los requisitos al comienzo
- Permite la entrega temprana de partes operativas de software.
- Las entregas facilitan la realimentación de las siguientes.

Se recomienda cuando los requisitos o el diseño no están completamente definidos y es posible que haya grandes cambios.

Los procesos ágiles se basan en este modelo, con iteraciones cortas.

5.3. Modelos de desarrollo Ágil

La visión ágil del desarrollo es una respuesta a los fracasos y las frustraciones del modelo en cascada. Contempla un enfoque para la toma de decisiones y la forma de organización en los proyectos de software, basándose en los modelos de desarrollo iterativo e incremental, con iteraciones cortas (semanas) y sin que dentro de cada iteración tenga que haber fases lineales.

En el 2001, 17 representantes de las nuevas tecnologías y el desarrollo de software se juntaron para discutir sobre el desarrollo de software. De esa reunión surgió el Manifiesto Ágil:

“Estamos descubriendo mejores maneras de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de esta experiencia hemos aprendido a valorar:

- **Individuos e interacciones** sobre procesos y herramientas
- **Software funcionando** sobre documentación extensiva
- **Colaboración con el cliente** sobre negociación contractual
- **Respuesta ante el cambio** sobre seguir un plan

Esto es, aunque los elementos a la derecha tienen valor, nosotros valoramos por encima de ellos los que están a la izquierda.”

Estas ideas se explican en los 12 principios para el desarrollo ágil de software.

¿En qué consiste el desarrollo ágil? Los 12 principios.

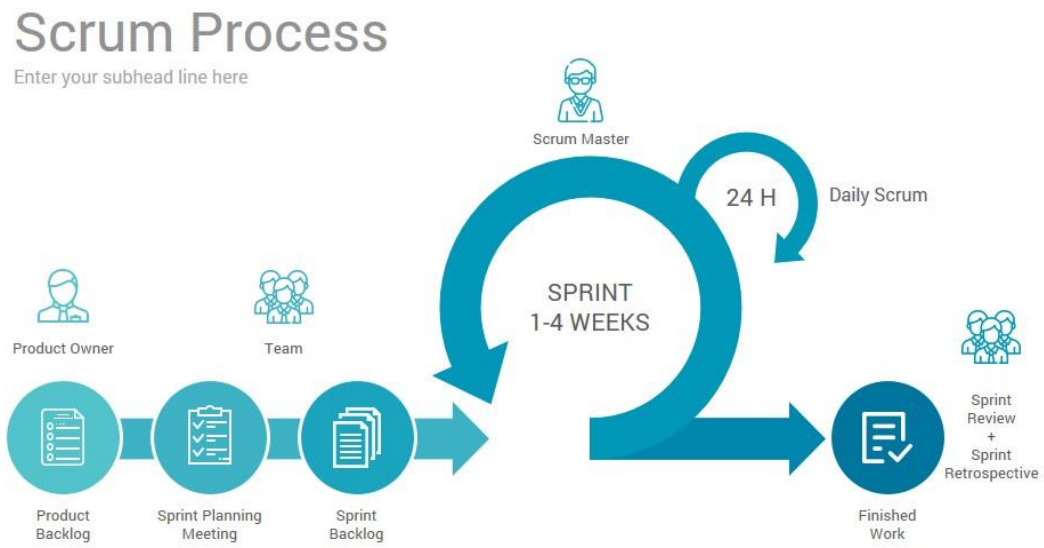
1. Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.

2. Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
3. Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
4. Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.
5. Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
6. El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
7. El software funcionando es la medida principal de progreso.
8. Los procesos Ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
9. La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.
10. La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
11. Las mejores arquitecturas, requisitos y diseños emergen de equipos autoorganizados.
12. A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

El abanico de metodologías ágiles es amplio, existiendo métodos para organizar equipos o proyectos, y técnicas para escribir y mantener el software. Tanto las técnicas de desarrollo de software como las de organización de proyectos se pueden combinar siendo de las más famosas la programación extrema junto con la gestión Scrum.

5.3.1. SCRUM

Es una técnica de *gestión de proyectos* englobada dentro del paradigma de desarrollo ágil. No se aplica únicamente al desarrollo de software, sino a la gestión de cualquier tipo de proyectos. Como las demás, se basa en el desarrollo **iterativo e incremental** en el que el proyecto se planifica en diversos bloques temporales (un mes, normalmente) llamados **Sprints**. Cumple con las características del desarrollo ágil. La metodología Scrum se organiza de la siguiente forma:



FASES

- Reunión inicial

El cliente nos indica qué es lo que quiere. De esta reunión entre el cliente y el *Scrum Master* surge la lista de requisito del producto (*Product Backlog*). Se realiza una sola vez al inicio del proyecto, a diferencia de las demás fases que se realizan en cada Sprint.

Planificación de la iteración (**Sprint Planning**)

Es una reunión en la se habla con el cliente quien indica los requisitos prioritarios de la iteración (Sprint), se plantean las dudas y se traducen dichos requisitos a tareas (Sprint Backlog) que deben ser realizadas por cada uno de los miembros del equipo de desarrollo durante el Sprint actual. Estas tareas se incorporan a un panel que controlará su estado de realización (por hacer, en progreso, terminado, pendiente, etc).

Ejecución de la iteración (**Sprint**)

Es el bloque de tiempo, iteración, en el cuál se realizan la tareas (2-4 semanas), siendo recomendable que siempre tengan la misma duración y la duración esté definida por el grupo en base a su experiencia.

Reunión diaria del equipo (**Daily Scrum Meeting**)

Cada día se celebra una reunión rápida (no más de 10 minutos, siempre puntual y suele ser de pié) en la que se habla del estado de las tareas del Sprint (Sprint Backlog), y en el panel en el que las tenemos indicadas las marcamos como terminadas, bloqueadas, en progreso, etc. Solo se habla del estado de las cosas y solo los involucrados pueden hablar.

Revisión de la entrega (**Sprint Review**)

Es una reunión que se produce al final del Sprint en la que se presenta al cliente el incremento (parte del producto pactada) realizado durante el Sprint. En ella el cliente puede ver cómo han sido desarrollados los requisitos que indicó para el Sprint, si se cumplen las expectativas, y entender mejor qué es lo que necesita.

- Retrospectiva (**Sprint Retrospective**)

Cuando se ha terminado un *Sprint*, se comprobarán, medirán y discutirán los resultados obtenidos en el periodo anterior a través de las impresiones de todos los miembros del equipo. El propósito de la retrospectiva es realizar una mejora continua del proceso.

ROLES

- Cliente (**Producto Owner**)

Es quién plantea los objetivos y requisitos y ayuda al usuario a escribir las **historias de usuario** que se incorporarán al *Sprint Backlog*, definiendo las prioridades. Representa a todas las personas interesadas en el proyecto (usuarios finales, promotores, etc). Es quién indica qué quiere en total y qué quiere en cada Sprint.

- Facilitador (**Scrum Master**)

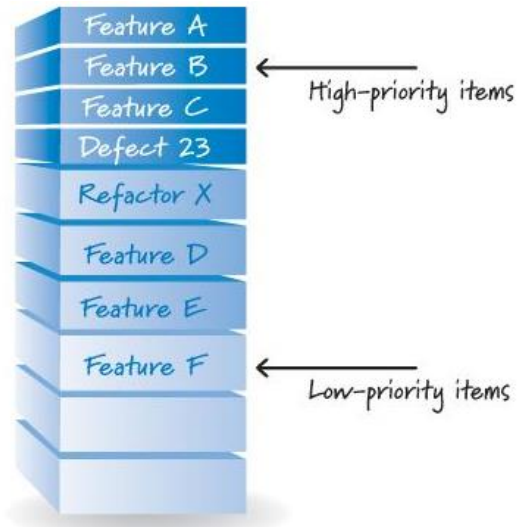
Se asegura de que se sigan los valores y principios ágiles, de la existencia de las listas de requisitos para cada iteración, facilita las reuniones, quitar los impedimentos que se van encontrando en el camino y proteger y aislar al equipo de interrupciones.

- Equipo (**Team**)

Tienen un objetivo común, se auto-organizan, comparten la responsabilidad del trabajo que realizan, y cuyos miembros confían entre ellos. 5-9 personas es lo ideal, aunque si hay más se deben hacer más equipos

HERRAMIENTAS

- Lista de requisitos priorizada (**Product Backlog**)



Es un documento abierto que representa la lista de objetivos o requisitos, y expectativas del cliente respecto a los objetivos y entregas del producto. Solo puede ser modificado por el product owner (cliente) aunque con ayuda del equipo y el “scrum master”, quienes proporcionan el coste estimado tanto del valor de negocio como del esfuerzo de desarrollo. Representa el qué va a ser construido en su totalidad.

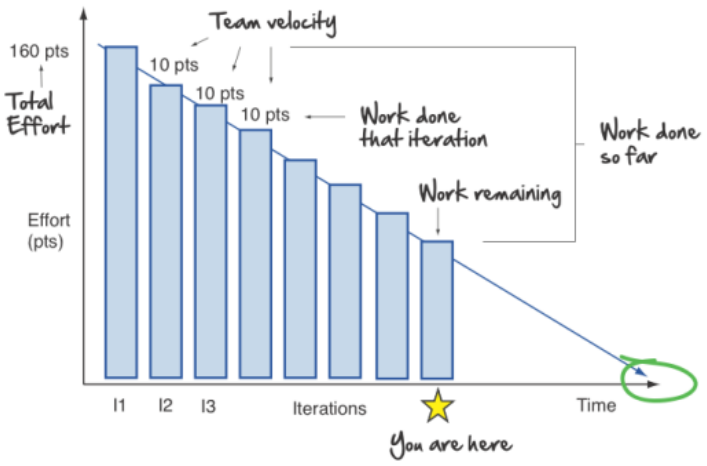
- Lista de tareas pendientes del Sprint (**Sprint Backlog**)

Lista de tareas que el equipo elabora en la reunión al inicio del Sprint,Sprint Planning, como plan para completar los requisitos para la iteración. Los requisitos tienen que estár claramente detallados en tareas (historias de usuario), a diferencia del Product Backlog. No se deben agregar requisitos al Sprint Backlog a menos que su falta amenace al éxito del proyecto. Representa el cómo el equipo va a implementar los requisitos.

ID	Status	Task	Task Type	Priority Lvl.	Est. Time (hours)	Actual Time (hours)	Owner of Task
1	Complete	Finish project proposal and decide on a topic	Planning	Very High	1	1.5	Jennifer
2	Complete	Set up MySQL and PhpMyAdmin	Content	Very High	0.5	1	Jennifer
3	Complete	Create online portfolio using Wordpress	Content	Very High	0.5	0.5	Jennifer
4	Complete	Work on website wireframe and layout	Design	Very High	2	0.5	Jennifer
5	Complete	Code HTML/CSS for the pages	Feature	Very High	4	3	Jennifer
6	Complete	Decide what pages must be created for the website	Planning	High	1	0.5	Jennifer
7	Complete	Create and begin formating the other pages	Feature	High	8	3	Jennifer
8	Complete	Code some sort of navigation so users can navigate through the different pages	Feature	High	2	1	Jennifer
9	In Progress	Do research on my topic of clean drinking water	Content	High	10	TBA	Jennifer
10	In Progress	Add information that I've gathered in my research to my website	Content	Medium	2	TBA	Jennifer
11	In Progress	Integrate a 3rd party service such as Google Maps	Feature	Medium	1.5	TBA	Jennifer
12	In Progress	Find / take own pictures to use	Design	Low	1	TBA	Jennifer
13	In Progress	Find / create video(s) to incorporate	Feature	Low	1	TBA	Jennifer
14	In Progress	Create graphics and visuals using Photoshop	Design	Low	1.5	TBA	Jennifer
15	In Progress	Look into incorporating audio to the website	Feature	Low	1	TBA	Jennifer

- Gráficos de trabajo pendiente (**Burndown Chart**)

Es un gráfico de trabajo pendiente a lo largo del tiempo, en el que se muestra la velocidad a la que se están completando los requisitos. También ayuda a intuir si el equipo terminará en el tiempo estimado. Lo normal es que la línea que une todos los sprints completados sea descendente, pero s se añaden nuevos requisitos durante el proceso, puede ser ascendente.

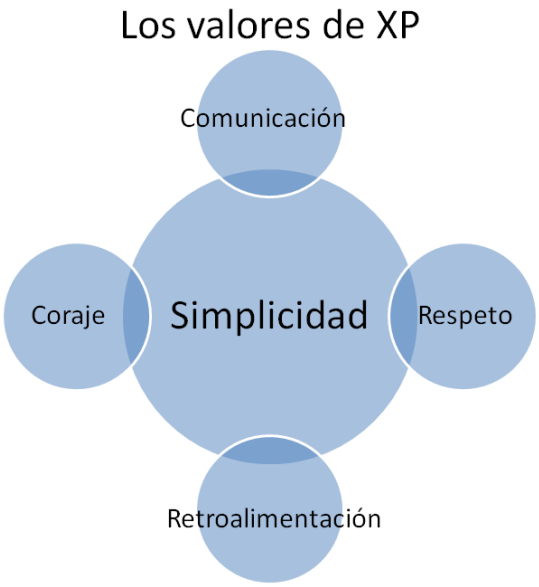


Scrum permite conocer en un vistazo rápido cual es el estado general de las cosas, detectando rápidamente qué tareas se han quedado atascadas o qué equipos no están rindiendo al nivel que se esperaba. Es un método de desarrollo ágil ideal para entornos con mucha incertidumbre en cuanto al trabajo a realizar, en los que las tareas cambian muy rápidamente y son susceptibles de olvidarse.

5.3.2. Programación Extrema

Conocida comúnmente como **XP** (eXtreme Programming), es una técnica de implementación de software formulada por Kent Beck, uno de los impulsores del manifiesto ágil. Es una metodología que hace énfasis en la adaptación continua a cambios en el plan, antes que en seguir un plan firme. Se considera que los cambios en los resquicitos deben ser bienvenidos ya que ayudan a crear un programa más sólido. Es una metodología enfocada en grupos de desarrollo pequeños en los que hay gran colaboración con el cliente.

Para conseguirlo, se diseña una forma de trabajar fundamentada en un modelo de desarrollo incremental con **entregas extremadamente rápidas** y esperando feedback (opinión, información, cómo lo quiere) del cliente casi a diario por parte del cliente. En cada una de las iteraciones del desarrollo se programa reflexionando, diseñando y documentando el código, a medida que se escribe.



- **Simplicidad:**

El código debe ser lo más claro y sencillo, empleando la refactorización como medida de crear “código limpio” a medida que crece. Del mismo modo es más fácil crear la documentación de un código simple en el que cada método tiene una clara función y se detalla con pocas palabras. Elegir los nombres apropiados para las variables métodos y clases ayuda en esta tarea.

- **Comunicación:**

Es muy habitual realizar programación por parejas, por lo que la comunicación entre desarrolladores es muy importante. La mejor forma de comunicarse es a través del código; si es complejo, debemos esforzarnos en hacerlo más claro. El código autodocumentado es la forma más fiable de documentar. Solo debemos documentar aquello que sabemos que no va a cambiar, como Clases o la funcionalidad de un método.

- **Retroalimentación (Feedback):**

Como se realizan iteraciones muy cortas en las cuales se muestran los resultados al cliente, se minimiza el riesgo de que el programa creado no cumpla con los requisitos del cliente. La comunicación con el cliente es la mejor forma de obtener una opinión real sobre el estado del proyecto.

Además se realizan pruebas unitarias diarias que nos informan del “estado de salud” del código.

- **Coraje**

La programación extrema implica aceptar retos y arriesgarse. Debemos borrar código obsoleto sin importar el esfuerzo realizado, reconstruir el código constantemente, o incluso modificarlo si con ello los cambios futuros serán mas fáciles. Diseñar y programar para hoy, para no perder demasiado tiempo en el diseño y alargar la implementación.

- **Respeto**

Los miembros del equipo muestran respeto mutuo y valoran el trabajo de los demás componentes; esto mejora la autoestima, la motivación y por lo tanto la producción.

Características:

- ✓ Realización diaria de pruebas unitarias automatizadas.
- ✓ Entregas al cliente muy rápidas, cada pocos días.
- ✓ Programación por parejas.
- ✓ Integración equipo-cliente.
- ✓ Refactorización constante del código.
- ✓ Simplicidad del código.

Programación por parejas en la Programación Extrema:

Es una técnica de desarrollo ágil de software en la que dos programadores trabajan juntos en el mismo ordenador. Uno, el **controlador** (*o driver*), escribe el código mientras que el otro, el **observador**, revisa cada línea de código mientras que es escrita. Los dos programadores cambian su rol frecuentemente.

Mientras que el código es revisado por el observador, este va pensando en la dirección estratégica a tomar, incluyendo nuevas ideas para mejoras futuras y anticipándose a futuros problemas. De esta forma el controlador puede prestar su completa atención a terminar la tarea actual, usando al controlador como guía.

Características:

- Calidad del Diseño: programas más cortos, diseños mejores, pocos defectos, etc. El código debe ser comprensible para dos programadores, y se plantean más alternativas.
- Coste reducido de desarrollo: Cuando los defectos son la parte más cara del desarrollo de software, especialmente si se encuentran en etapas avanzadas, se minimizan programando por parejas.
- Aprendizaje y formación: el conocimiento pasa fácilmente entre programadores.
- Superación de problemas dificiles
- Incremento de la disciplina y mejor gestión del tiempo: menos distracciones para cada uno, menos tiempo perdido en tareas ajenas, etc.
- Menos interrupciones externas: cuesta más interrumpir a una pareja que a una persona que trabaja sola.