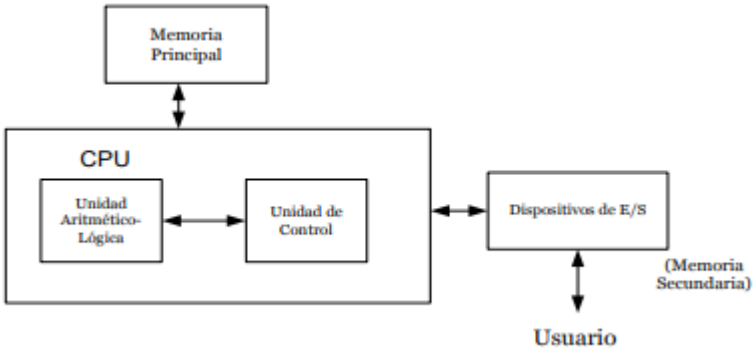
REPASO DE PREPARACIÓN DE EXAMEN DE ENTORNOS DE DESARROLLO (UNIDAD 1)

INTRODUCCIÓN AL DESARROLLO DEL SOFTWARE

HARDWARE

- Es la parte tangible (equipamiento físico)
- Generalmente un ordenador digital se basa en la arquitectura de Von Neumann (1945), la cuál divide el ordenador en las siguientes partes:
    - **Unidad de control**: Controla la ejecución de las operaciones y dirige el funcionamiento de todos los demás componentes de tal forma que el trabajo conjunto de todos conduzca a la consecución de las tareas específicas programadas en el sistema.



    - **Unidad aritmético-lógica**: Es la encargada de realizar operaciones aritméticas y lógicas. También puede realizar funciones más complejas como: raíces, funciones trigonométricas, etc.

> ***Al conjunto de UC y ALU, se le conoce como CPU (Central Process Unit). Así tenemos procesadores Intel, Motorola, AMD, etc.***

- **Memoria Principal**: Memorial de almacenamiento interno que opera a gran velocidad y almacena las instrucciones junto con los datos sobres los que actúan los programas. Está formada por celdas que son posiciones de memoria.
- **Dispositivos de I/O**: Son los que facilitan la interacción del usuario con la máquina.
- El ciclo de una instrucción es:
    1. La UC indica a la RAM la siguiente instrucción a ejecutar.
    2. La UC recibe la instrucción, procediendo a su análisis para determinar los operandos sobre los que actúa y su localización.
    3. Bajo las directrices de la UC, la ALU realiza la operación y se guarda el resultado en su destino.
    4. Una vez ejecutada la instrucción se incrementa el contador de programa y se pasa a realizar la ejecución de la siguiente instrucción.

GENERACIONES DE LOS ORDENADORES SEGÚN EL HARDWARE

Primera generación

- 1945-1955
- Tubos de vacío
- Se programaba en binario
- Se requería la intervención humana

Segunda generación

- 1955-1965
- Transistores
- Se utilizaba el lenguaje ensamblador
- Programación en lotes.

Tercera generación
- 1965-1971
- Circuitos integrados
- Se utilizaban lenguajes de alto nivel
- Se desarrolló la multiprogramación y los sistemas compartidos

Cuarta generación
- 1971-1985
- Circuitos integrados de alta densidad LCI
- Lenguajes de alto nivel
- Aparecen los ordenadores personales

Quinta generación
- 1985-actualidad
- Circuitos integrados de ultra alta densidad ULCI
- Se utilizaban lenguajes de alto nivel
- Computación en red, distribuida y auge del desarrollo del software

SOFTWARE

- Es la parte intangible o equipamiento lógico
- Es el encargado de comunicarse con el software (traducir órdenes comprensibles por el hardware)
- El concepto se utiliza por primera vez por Charles Babbage (1822).
- Alan Turing (1912-1954) profundizó en el concepto del software y destacó como criptógrafo por su máquina. Actualmente considerado como el padre la computación ya que diseñó el sistema de programación del primer ordenador comercial y sentó las bases de las IA.

CARACTERÍSTICAS

1. Lógico
2. Se desarrolla, no se fabrica.
3. No se estropea, una copia.
4. Se puede construir a medida, puede construirse a medida o enlatado.

TIPOS

Según sus funciones, podemos destacar:
- **Software de sistema**: Interactúa con el resto de programas y, además, controla el hardware. Podemos dividirlos en: SO, controladores y programas utilitarios.
- **Software de aplicación**: Programas con una finalidad más o menos concreta. Estos han sido diseñados para facilitar al usuario la realización de ciertas tareas, ej.: Ofimática, educativo, editores, etc.
- **Software de programación**: Conjunto de herramientas que permiten el desarrollo de programas informáticos, ej.: Compiladores, intérpretes, ensambladores, enlazadores.

Según su distribución:
- **Shareware**: Modalidad de distribución para que se pueda evaluar de forma gratuita durante un tiempo especificado.
- **Freeware**: Se distribuye sin cargo. A veces incluso incluye el código fuente, pero no es habitual.
- **Adware**: Shaeware que descargan publicidad. Al comprar la licencia se elimina esta publicidad.

LENGUAJES DE PROGRAMACIÓN

CONCEPTO

Según la RAE:

> ***Un lenguaje de programación es el lenguaje que facilita la comunicación con un computador mediante signos convencionales cercanos a los de un lenguaje natural. Es decir, un lenguaje de programación es una herramienta que nos permite crear programas ejecutables en un ordenador.***

CARACTERÍSTICAS

- **Facilidad de lectura y comprensión**: Los programas deben ser corregidos y modificados, por lo tanto, el lenguaje de programación debe facilitar la comprensión de los programas una vez escritos.
- **Facilidad de codificación**: Debe ser simple utilizar el lenguaje para desarrollar programas que resuelvan el tipo de problemas al que está orientado.
- **Fiabilidad**: Es fiable si funciona en **cualquier condición**. Depende de la comprobación de tipos de datos, del control de excepciones y del manejo de la memoria.
- **Posibilidad de usar un entorno de programación**: Se refiere a la existencia de herramientas para el desarrollo.
- **Portabilidad**: Un mismo programa debe funcionar en ordenadores diferentes.
- **Coste**: Relacionado con la curva del aprendizaje, codificación, ejecución, fiabilidad y mantenimiento.
- **Detallabilidad**: Define el número de pasos que es necesario definir en un programa para dar solución a un determinado problema.
- **Generalidad**: Indica las opciones de uso que tiene un lenguaje para resolver problemas de distinto tipo. Cuantos más problemas diferentes pueda solucionar más general es el lenguaje.

ELEMENTOS

Son 3:

Nivel léxico: Son símbolos que se pueden usar en el lenguaje:

- **Identificadores**: Nombres simbólicos que se da a ciertos elementos de programación, ej.: nombre de variables, tipos, etc.
- **Constantes**: Una constante es un dato cuyo valor permanece inalterable durante toda la ejecución del programa.
- **Operadores**: Símbolos que representan operaciones entre variables y constantes. Los operadores más comunes son: aritméticos, lógicos, incrementales, relaciones y de asignación.
- **Palabras claves**: Son aquellas que palabras predefinidas por el lenguaje que tienen un significado especial para este.
- **Comentarios**: Símbolo que se utiliza para documentar los programas explicando el código fuente. (cosa que nadie hace porque hasta desarrolladores dedicados piensan que tienes el conocimiento absoluto del sistema y aplicaciones que ellos hacen por lo cual esto es algo que solo existe en sueños).
- **Separadores**: Los separadores están formados por espacios en blanco, tabuladores y saltos de línea, y su función es ayudar al compilador a descomponer el programa en token y facilitar la legibilidad del programa.

Nivel sintáctico

Es el conjunto de normas que determinan cómo se pueden utilizar los símbolos del lenguaje para crear sentencias. Las reglas sintácticas pueden tener elementos terminales (símbolos léxicos) y elementos no terminales (construcciones gramaticales intermedias).

Una vez definida la sintaxis del lenguaje, el compilador determinará si las sentencias de un código fuente son válidas, de acuerdo a esta sintaxis. Para esto se utilizarán árboles de análisis sintáctico y algoritmos de parsing.

Nivel semántico;

La semántica de un lenguaje se refiere al significado que adoptan las distintas sentencias, expresiones y enunciados de un programa. La semántica engloba aspectos sensibles al contexto. Los principales elementos de la semántica son:

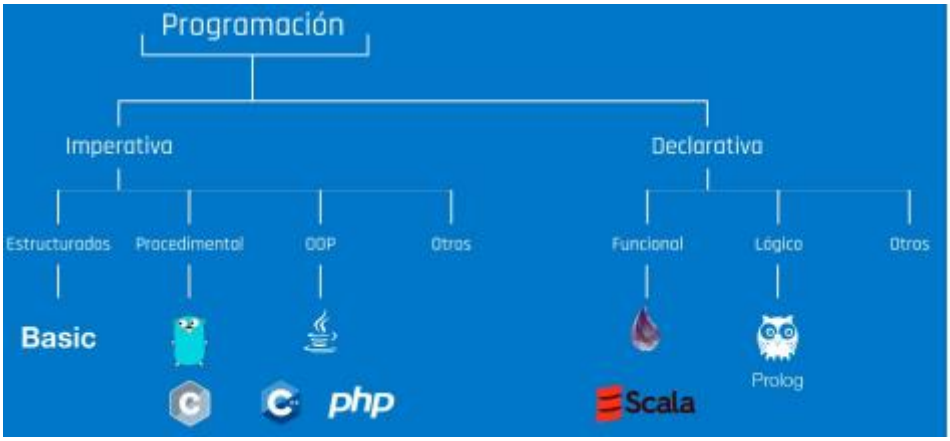
- **Variable**: Una variable representa un espacio de memoria para almacenar un valor de un determinado tipo. El valor de una variable, puede cambiar durante la ejecución del programa.
- **Valores y referencias**: Los valores son el estado de determinada celda o grupo de celdas de la memoria, mientras que las referencias indican la posición de esa celda en memorial. Estos dos

conceptos están muy involucrados con los punteros. La mayoría de los lenguajes los soportan, pero son una gran fuente de errores de programación.

- **Expresiones**: Son construcciones sintácticas que permiten combinar valores con operadores y producir nuevos valores. Las expresiones pueden ser aritméticas, relaciones, lógicas y condicionales. Cada una de estas tiene una semántica específica que la define.
- **Gramática de atributos**: Las gramáticas de atributos permiten formalizar aspectos sensibles al contexto, ej.: el chequeo de tipos depende del contexto porque debemos saber el tipo esperado y el actual y, determinar si son compatibles. El tipo esperado lo obtenemos del contexto analizando la definición de la variable.

PARADIGMAS

Un paradigma es un estilo de desarrollo de programa. Un lenguaje puede pertenecer a más de un paradigma a partir del tipo de ordenes que permite implementar, algo que tiene una relación directa con su sintaxis.



TIPOS:

Pueden ser por:

- **Nivel de abstracción**: es el nivel de proximidad que está el lenguaje de programación al máquina.
- **Clasificación cronológica**: por fecha de creación
- **Por paradigmas**: los paradigmas de programación consisten en el método que llevan a cabo los cálculos y la forma que deben estructurarse y organizarse las tareas que realiza el programa.
    - **Imperativa**: Es aquella en la que se describe paso a paso un conjunto de instrucciones que deben realizarse para ejecutarse para variar es estado del programa y hallar la solución.
    - **Declarativa**: Las sentencias describen el problema que se quiere solucionar; se programa diciendo lo que se quiere resolver a nivel de usuario, pero no las instrucciones necesarias para realizarlo.

Por nivel de abstracción:

- **Lenguaje máquina**: 0 y 1 que conforman operaciones que la máquina pueda entender.
    - Depende de la arquitectura de la máquina, CPU, la cual implica baja o nula portabilidad.
    - No existen comentarios.
    - Datos referenciados a dirección de memoria específica.
    - Operaciones simples
    - Código muy rígido, lo cuál conlleva a poca versatilidad a la hora de hacerse.
- **Lenguaje ensamblador o bajo nivel**: Los 0 y 1 se reemplazan por símbolos reconocibles llamados **mnemotécnicos** (normalmente abreviaturas en inglés) para representar las instrucciones.
    - Permite la utilización del direccionamiento simbólico, es decir, en vez de utilizar direcciones binarias podemos identificar los datos con nombres (variables).
    - Permite comentarios
    - Sigue dependiendo de la arquitectura
    - Realiza una gestión óptima de los recursos de hardware, siendo muy eficiente.
    - Es necesario traducirlo es código a lenguaje máquina con un ensamblador.

- **Lenguaje de alto nivel**: Permiten programar sin necesidad de conocer el funcionamiento interno de la máquina. Normalmente una instrucción de estos lenguajes equivale a varias instrucciones máquina. Se diseñaron con la finalidad de que las personas escriban y entiendan los programadas de un modo más fácil que los lenguajes máquina y ensambladores.
    - Son independientes de la arquitectura por lo cual son portables
    - Las operaciones se expresan de una forma muy similar al lenguaje matemático o lenguaje natural (generalmente inglés).
    - Permite el uso de los comentarios y de las variables.
    - Requieren de un compilador para su traducción al lenguaje máquina.

Por generación (lenguajes de programación):

- **Primera generación**: Coincide con el lenguaje máquina visto en la clasificación anterior.
- **Segunda generación**: Lenguaje ensamblador.
- **Tercera generación**: Aparecen lenguajes de alto nivel con técnicas de programación (estructurada, concurrente, orientada a objetos) que logran un gran rendimiento computacional con respecto a los anteriores.
- **Cuarte generación**: Aparición de herramientas orientadas al desarrollo de aplicaciones de gestión que permiten automatizar operaciones como, definir BBDD, informes, consultas…
- **Quinta generación**: IA. Se caracterizan por lenguajes de programación declarativa y lógica.

Por paradigma:

- **Programación estructurada**: Orienta a mejorar la claridad, calidad y tiempo de desarrollo de un programa recurriendo a subrutinas y tres estructuras básicas: secuencia, selección e iteración.
- **Programación modular**: Consiste en dividir el programa en módulos o subprogramas con el fin de hacerlo más legible y manejable.
- **Programación orientada a objetos**: Está basada en la programación imperativa pero el núcleo central de este paradigma es la unión de los datos y el procesamiento en una entidad llamada "objeto", la cuál a su vez es relacionable con otras entidades " objeto".
- **Programación dinámica**: Se basa en dividir un problema en más pequeños para analizarlos y resolverlos de forma más cercada al óptimo. Este paradigma se basa en el modo de realizar los algoritmos, por lo que se pueden usar cualquier lenguaje imperativo.
- **Programación funcional**: Se basa en la definición de predicados y es de corte más matemático.
- **Programación lógica**: Se basa en la definición de relaciones lógicas, son lenguajes en los que se especifican un conjunto de hechos y una serie de reglas que permiten la deducción de otros hechos. Esto permite al sistema encontrar la solución.
- **Programación con restricciones**: similar al anterior, pero usando ecuaciones.
- **Lenguaje específico del dominio o DLS**: Se denomina así a los lenguajes desarrollados para resolver un problema específico, pudiendo entrar dentro de cualquier grupo anterior.
- **Programación multiparadigma**.

COMPILADORES

CLASIFICACIÓN DE LOS LENGUAJES SEGÚN SU EJECUCIÓN

- Compilado: se traduce el programa escrito en un determinado lenguaje (código fuente) a un lenguaje máquina (código objeto). Si el código fuente tiene errores el compilado devolverá errores y no se generará el fichero objeto.
- Interpretado: Cada línea se va traduciendo sin generar un fichero compilado (objeto).

Un lenguaje compilado suele ser más rápido que un lenguaje interpretado, pero cada vez que se realiza un cambio en este, tiene que volver a compilarse.

JAVA

Lenguaje compilado creado en 1991.

Permite:

- Aplicaciones independientes, **Stand-alone Application**.
- Aplicación ejecutada en el navegador, **Applet.**
- Aplicación ejecutada en el servidor, **Servlet**.

Para poder desarrollar aplicaciones Java, necesitamos un JDK, **Java Development Kit**.

Para poder ejecutar el código Java, nos hará falta un JRE, **Java Runtime Envioment**.

El código java es interpretado por la JVM. Esta genera un código intermedio llamado **bytecode** (híbrido entre lenguaje de alto nivel y ensamblador).

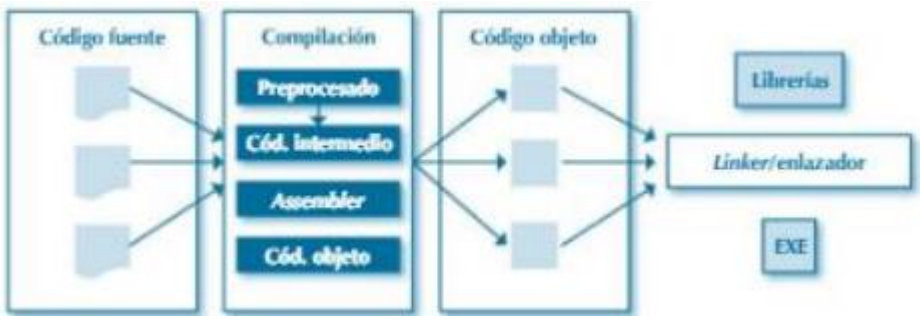
CODIFICACIÓN

Los traductores son programas cuya finalidad es traducir lenguajes de alto nivel a bajo nivel.

Un interprete traduce el código línea a línea, tiene que estar en memoria para poder realizar este proceso.

Un compilador traduce el código fuente a máquina. El compilador está en la máquina de desarrollo. El código generado solo funcionará en una máquina con un hardware y software determinado. Si se cambian de hardware o software, habría que recompilarlo.

Fases de un compilador:



1. **Edición**: Aquí el programa fuente debe ser tecleado a la computadora con un EDITOR (código fuente)
2. **Compilación**: El compilador se encarga de traducir la edición del programa fuente a lenguaje máquina. De ser necesario la compilación se repite hasta no producir errores, dando como resultado el programa objeto. → Programa objeto (código objeto).
3. **Enlace**: el OS es instruido a tomar el programa objeto y ligarlo con las librerías del programa compilador, dando como resultado un programa ejecutable.
4. **Ejecución**: con el ejecutable listo, la aplicación puede "correr" en el SO obteniendo por salida los resultados del programa.

La compilación

1. **Código fuente**.
2. **Análisis lexicográfico**: Se busca dentro del código fuente palabras reservadas, operaciones, caracteres de puntuación y se agrupan en cadenas denominadas **lexemas**.
3. **Análisis sintáctico semántico**: Agrupa los resultados anteriormente obtenidos como frases gramaticales. Tras el análisis sintáctico, verifica la coherencia de las frases gramaticales, es decir: si los tipos de datos son correctos, si los *arrays* tienen el tamaño y tipo adecuados, y así consecutivamente con todas las reglas semánticas de nuestro lenguaje.
4. **Generador de código intermedio.**
5. **Código intermedio**: Tras el análisis anterior, se genera una representación intermedia en psudoensamblador, facilita las tareas de las traducción restante.
6. **Optimizar código**: Optimiza el código anterior para que sea más eficiente en la máquina. Ej.: Crear variables de intercambio de valores en decisiones.
7. **Código optimizado**: Tras esta optimización de los recursos, genera un código optimizado.
8. **Código objeto**: Genera el código objeto con posiciones de memoria sin establecer, en otras palabras, no sabemos en que zona de la memoria se ejecutará nuestro programa.
9. **Librerías→Enlazador.**
10. **Código ejecutable.**

DEPURACIÓN

Aunque el código compile, tenemos que realizar una serie de pruebas poniendo nuestro programa al límite con la finalidad de cerciorar su funcionamiento y evitar errores en su operación (**ERRORES DE EJECUCIÓN**).

Las pruebas pueden ser:

- Valores extremos de entrada.
- Realizar operaciones manualmente y compararlas con el resultado del programa.
- Ingresar datos erróneos para determinar su fiabilidad, es decir, su capacidad de recuperarse o, en su caso, el nivel de control frente al uso inadecuado del programa.

La mayoría de errores que no detecta el programa al compilarse son los de **ejecución y los lógicos**.

- Los de ejecución: se dice que la computadores los puede "entender" pero no ejecutar como: dividir entre cero, raíz cuadrada de un número negativo, realizar una operación con un dato leído como carácter en vez de número, etc.
- Los lógicos: son más complicados de corregir ya que generalmente son producto de mal análisis y mal diseño de programa.

TÉCNICAS DE DESARROLLO GRÁFICO.

- Diagrama de contexto nivel 0:
    - Representan todas las interacciones que realiza un sistema con su entorno.
    - Se dibuja un solo proceso que representa al sistema en cuestión y se escribe su nombre en dicha burbuja.
    - De el solamente parten los flujos de datos que denotan las interrelaciones entre el sistema y sus agentes externos, no admitiéndose otros procesos ni almacenamiento de datos. Esto es debido a que son procesos estructurados y ordenados que poseen una cardinalidad.
- Diagrama de nivel superior: nivel 1:
    - Se plasman todos los procesos que describen al proceso principal.
- Diagrama de detalle o expansión: Nivel 2:
    - En estos se explotarán las excepciones a los caminos principales de la información dado a que aumentan progresivamente el nivel de detalle.

INTRODUCCIÓN A LA INGENIERÍA DEL SOFTWARE

HISTORIA DEL SOFTWARE

1930-50

Desde la década de '30 hasta '50, las tarjetas perforadas se convirtieron en la fuerza motriz de las empresas, ya que se utilizaron prácticamente todas las máquinas de contabilidad de oficina. Las tarjetas fueron creadas con lenguajes de programación como FORTRAN de IBM y COBOL del departamento de defensa de USA.

- Proyecto ENIAC

1950-60

El término "Software" se creó a finales de '50 y pronto fue adoptado por toda la industria. Dividiendo el software en dos tipos principales: Software de sistema y programa aplicaciones.

El de sistema incluye los procesos generales de la ejecución del programa, tales como compiladores y sistema operativo de disco.

Aplicaciones del programa como pueden ser aplicaciones de oficina.

- Mainframe IBM 1401 (1959).

1960-80

A finales de los 60, la potencia de las maquinas empezó a aumentar de forma considerable.

Empezaron a aparecer los lenguajes de programación de alto nivel., y las máquinas programas mucho más complejos de los desarrollados hasta la época. En definitiva, fue un salto tremendo en cuanto al potencial de hardware, que no fue acompañado por un salto en el desarrollo de software.

En esta época, se empezó a concebir el software como un producto y se empezaron a desarrollar algunos proyectos para que funcionaran en las máquinas de la época.

Pero aparecieron importantes problemas:

- Los productos excedían la estimación de costes.
- Había retrasos en las entregas.
- Las prestaciones no eran las solicitadas
- El mantenimiento se hacía extremadamente complicado y a veces imposible, las modificaciones tenían un coste prohibitivo…

CRISIS DEL SOFTWARE.

Fue acuñada a principios de los años 70, cuando la ingeniería de software era prácticamente inexistente.

El término expresaba las dificultades del desarrollo de software frente al rápido crecimiento de la demanda por software, de la complejidad de los problemas a ser resueltos y de la inexistencia de técnicas establecidas para el desarrollo de sistemas que funcionaran adecuadamente o pudieran ser validados.

- Historia del Altair 8800 (1974).

Una de las principales causas de esto, si no la principal, era el enfoque dado al proceso del desarrollo de software, el cual era malo e incluso a veces era inexistente.

En este proceso, solo ¼ del tiempo de desarrollo se dedicaba a las fases de análisis, diseño, codificación y pruebas, y más de ¾ del tiempo se dedicaba a correcciones y mantenimiento.

Es evidente que dedicándole solo ¼ del tiempo a las primeras fases , se arrastraran errores graves, sobre todo procedente de las fases de análisis y diseño, lo que dificultaba muchísimo la implementación, produciendo constantes paradas y retrocesos para revisar este análisis/diseño.

EJEMPLOS DE PROYECTOS FALLIDOS.

Accidente de un F-18 (1986) (error de un giro descontrolado atribuido a una expresión if then donde no existía un else ya que fue considerado 'innecesario')

Muertes por el Therac-25 (1985-1987) causó muchas muertes por la cantidad de radiaciones aplicadas sin control que fue un error de software (radioterapia).

Sobrecoste, retraso y cancelación en el sistema del Bank of America (1988).

NACIMIENTO DE LA INGENIERÍA DEL SOFTWARE

En la primera conferencia sobre esto que fue allá por 1968 en Múnich, financiada por la OTAN. Allí donde se adoptó el término, hasta entonces prácticamente desconocido y quien lo usó por primera vez fue Fritz Bauer.

La **Ingeniería del software** es una disciplina que se ocupa de proporcionar un marco adecuado para la gestión de los proyectos software, y para el diseño de aplicaciones empresariales con una arquitectura software que favorezca su mantenimiento y evolución a lo largo del tiempo.

> Según estimaciones, el 26% de los grandes proyectos de software fracasan, el 48% deben modificarse drásticamente y sólo el 26% tienen rotundo éxito. La principal causa del fracaso de un proyecto es la falta de una buena planificación de las etapas y mala gestión de los pasos a seguir. ¿Por qué el porcentaje de fracaso es tan grande? ¿Por qué piensas que estas causas son tan determinantes?

INGENIERÍA DEL SOFTWARE

Estos facilitan la gestión del proceso de desarrollo y suministran a los desarrolladores bases para construir de forma productiva software de alta calidad.

La ingeniería del software abarca un conjunto de tres elementos clave:

- Métodos.
- Herramientas.
- Procedimientos.

MÉTODOS

Indican cómo construir técnicamente el software y abarcan una amplia serie de tareas que incluyen la planificación y estimación de proyectos, el análisis de requisitos, el diseño de estructuras de datos, programas y procedimientos, la codificación, las pruebas y el mantenimiento.

Los **métodos** introducen frecuentemente una notación específica para la tarea en cuestión y una serie de criterios de calidad.

Ej.: Cuestionarios, entrevistas, grupos de trabajo, …

HERRAMIENTAS

Las herramientas proporcionan un soporte automático o semiautomático para utilizar los métodos. Existen herramientas automatizadas para cada una de las fases vistas anteriormente, y sistemas que integran las herramientas de cada fase de forma que sirven para todo el proceso de desarrollo.

PROCEDIMIENTOS

Definen la secuencia en que se aplican los métodos, los documentos que se requieren, los controles que permiten asegurar la calidad y las directrices que permiten a los gestores evaluar los progresos.

En los procesos se distinguen 4 fases básicas:

- **Análisis**: Proceso de reunión de requisitos funcionales y no funcionales de un sistema.
- **Diseño**: Se refiere al establecimiento de las estructuras de datos, la arquitectura general de software, interfaz…
- **Implementación**: Traducimos el diseño en una forma legible por la máquina. Lo programamos.
- **Pruebas**: Una vez generado el software comienzan las pruebas. Para demostrar que no se encuentran fallos.

FASES DEL DESARROLLO DEL SOFTWARE.

El ciclo de vida: "Todas las etapas por las que pasa un proyecto de software desde que se inicia hasta que se finaliza y se considera una solución completa, correcta y estable.

1. Obtención de requerimientos
2. Análisis del sistema
3. Diseño del sistema
4. Implementación del sistema
5. Pruebas del sistemas
6. Instalación y mantenimiento del sistema.

Para empezar tenemos que preguntarnos lo siguiente:

¿Qué se va a desarrollar?

1. Estudio del sistema
2. Planificación del sistema
3. Especificación de requisitos
4. Análisis del sistema.

¿Cómo se va a desarrollar?

1. Diseño (arquitectura, estructura de datos, …)
2. Codificación e implementación
3. Pruebas

Cambios que se puede producir en el sistema

1. Adaptación o mejora del sistema.

AHORA SE EMPEZARÁN A EXPLICAR A DETALLE LAS FASES DEL DESARROLLO DE UNA APLICACIÓN PARA EVITAR ACCIDENTES DEL PASADO COMO EL F-18.

FASE INICIAL

En esta fase se hacen estimaciones de **viabilidad** del proyecto, es decir, si es rentable o no y se establecen las bases de las fases del proyecto.

ANÁLISIS

Se analiza el problema, es decir, se recopila, examina y formulan los requisitos del cliente.

Es fundamental, tener claro **QUÉ hay que hacer (comprender el problema)**.

Se realizan varias reuniones con el cliente, generando documentación en la que se recopilan los requisitos, Especificación de Requisitos del Software **(ERS)**.

Al finalizar estas reuniones, se genera un acuerdo en el que el cliente se compromete a no variar los requisitos hasta terminar una primera **release**.

Para facilitar la comunicación con el cliente, se utilizan técnicas como:

- Casos de uso (técnica **UML**, en la que se representan los requisitos funcionales del sistema.
- Brainstorming (desde distintos puntos de vista).
- Desarrollo conjunto de aplicaciones (**JAD**) y planificación conjunta de requisitos (**JRP**). Apoya en la dinámica de grupos y, respectivamente los productos generados comprenden los requisitos claves o estratégicos.
- Prototipos (versión inicial del sistema).

Hay dos tipos de requisitos:

- **Funcionales**: Describen con detalle la función que realiza el sistema
- **No funcionales**: Tratan de las características del sistema como fiabilidad, mantenibilidad, plataforma hardware, S.O, restricciones, etc.

Para su representación, se pueden utilizar:

- Diagrama de flujo de datos (**DFD**) de distintos niveles: Representan el flujo de datos entre los distintos procesos, entidades externas y almacenes que forman el sistema.
- Diagrama de flujo de control (**DFC**): Similares a los de DFD pero muestran el flujo de control.
- Diagramas de transición de estados (**DTE**): Representa cómo se comporta el sistema como consecuencia de sucesos externos.
- Diagramas Entidad/Relación (**DER**): Se usa para representar los datos y sus relaciones entre ellos.

Diseño

Consiste en dividir el problema en partes y se determina la función de cada una de estas. Se decide **CÓMO se hace (CÓMO se resuelve el problema)**.

Se elabora una documentación técnica y detallada de cada módulo del sistema.

La documentación es realizada por el ***analista junto al jefe del proyecto***.

Los tipos de diseños más utilizados son:

- **Diseño estructurado (la vieja confiable)**: Está basado en el flujo de datos a través del sistema
- **Diseño orientado a objetos**: El sistema se entiende como un conjunto de objetos los cuales tienen propiedades, comportamientos y se comunican entre ellos.

Codificación

El programador codificará el software según los criterios seleccionados en la etapa de diseño.

Al trabajar en equipo, debe haber unas normas de codificación y estilo, claras y homogéneas.

Cuanto más exhaustivo haya sido el análisis y el diseño, la tarea será más sencilla, pero nunca está exento de necesitar reanálisis o un rediseño si se encuentran problemas al programar.

En esta fase, se genera documentación del código que se desarrolla: entradas, librerías, etc.

Se pueden distinguir dos tipos:

- **Documentación del proceso**: Comprende las fases de desarrollo y mantenimiento.
- **Documentación del producto**: Describe el producto que se está desarrollando, se divide en dos tipos: Documentación del usuario y documentación del sistema.

Pruebas

Se realizan pruebas para verificar que el programa se ha realizado acuerdo las especificaciones originales, es decir, se comprueba lo que **DEBE HACER**.

Existen varios tipos de pruebas:

1. **Funcionales**: Se validan las especificaciones establecidas por el cliente y, posteriormente, será validad por este.
2. **Estructurales**: Se realizan pruebas técnicas sin necesidad del consentimiento del cliente. Se realizarán cargas reales de datos.

En esta fase comprende las siguientes tareas:

- **Verificación**: Se trata de comprobar si se está construyendo el software correctamente, es decir si implementa correctamente la función.
- **Validación**: Trata de comprobar si el software construido se ajusta a los requisitos del cliente.

En general, las pruebas las debería realizar personal diferente al que codificó, con una amplia experiencia en programación.

EXPLOTACIÓN

Consiste en la instalación y puesta en marcha del software en el entorno de trabajo del cliente, es decir, el software es instalado y utilizado en un entorno real de uso cotidiano.

Es la fase más larga, ya que suele conllevar múltiples incidencias (**bugs**)y nuevas necesidades.

En esta etapa se realizan las siguientes tareas: Estrategia para la implementación, pruebas de operación, uso operacional del sistema y soporte al usuario.

MANTENIMIENTO

Se mantiene el contacto con el cliente para actualizar y modificar la aplicación en el futuro.

En esta fase se puede incorporar código para soluciones a problemas, ampliar la funcionalidad para un cliente,…

Existen distinto tipos de mantenimiento:

- **Mantenimiento adaptativo**: Tiene como objetivo la modificación del software por los cambios que se produzcan, tanto en el hardware, como en el software den entorno en el que se ejecutan. Este mantenimiento es el más frecuente.
- **Mantenimiento correctivo**: Es muy probable que después de la entrega, el cliente encuentre fallos o defectos que deben ser corregidos.
- **Mantenimiento perfectivo**: Consiste en modificar el producto SW para incorporar nuevas funcionalidades y nuevas mejoras en el rendimiento del software.
- **Mantenimiento preventivo**: Consiste en modificar el producto sin alterar las especificaciones del mismo, con el fin de mejorar y facilitar las tareas de mantenimiento (por ejemplo: reestructurar programas para mejorar su legibilidad, añadir comentarios para facilitar su comprensión, etc.).

Un ciclo de vida es el conjunto de fases por las que software pasa desde que surge la idea hasta que muere.

De los ciclos de vida destacan ciertos aspectos clave que son:

- Ciclo de vida lineal o en cascada
- Ciclo de vida evolutivo

CICLO DE VIDA EN CASCADA

Ciclo de vida en el que las distintas fases se van encadenando las etapas de manera consecutiva. Es decir, la siguiente no empieza hasta que no haya terminado la anterior.

Este ciclo no comprendía la retroalimentación entre fases y la posibilidad de volver atrás.

Por lo tanto, al llevarlo a la práctica lo que ocasionó que no se supiera reaccionar a errores ocurridos en una fase avanzadas.

Sus ventajas son:

- Fácil de comprender, planificar y seguir.
- La calidad del producto resultante es alta.
- Permite trabajar con el personal poco cualificado.

Los inconvenientes:

- Necesidad de tener todos los requisitos definidos desde el principio (pueden surgir imprevistos).
- Es difícil volver atrás si se cometen errores en una etapa.
- El producto no está disponible para su uso hasta que no está totalmente terminado.

CASCADA CON RETROALIMENTACIÓN

Se recomienda (dada la necesidad de tener todos los requisitos desde el principio y la dificultad de volver atrás) cuando:

- El proyecto es similar a alguno que ya se haya realizado con éxito anteriormente
- Los requisitos son estables y están bien comprendidos.
- Los clientes no necesitan versiones intermedias.

CICLO DE VIDA V

El modelo V viene para corregir fallos del modelo en cascada incluyendo explícitamente la retroalimentación.

El problema es que los fallos detectados en fases tardías de un proyecto obligaban a empezar de nuevo, lo cual supone un gran sobrecoste.

CICLO DE VIDA INCREMENTAL

Este modelo busca reducir el riesgo que surge entre las necesidades del usuario y el producto final por malos entendidos durante la etapa de solicitud de requerimientos.

El ciclo se compone de iteraciones, las cuales, están compuestas por fases básicas.

Al final de cada iteración se le entrega al cliente una versión mejorada o con mayores funcionalidades del producto.

El cliente al finalizar cada iteración, evalúa el producto y lo corrige o propone mejoras.

Estas iteraciones se repetirán hasta que se desarrolle un producto que satisfaga las necesidades del cliente.

Sus ventajas son:

- No se necesitan conocer todos los requisitos desde el comienzo.
- Permite la entrega temprana al cliente de partes operativas del software.
- Permite separar la complejidad del proyecto, gracias a su desarrollo por parte de cada iteración o bloque.
- Las entregas facilitan la retroalimentación de los próximos entregables.

Sus desventajas:

- Es difícil estimar el esfuerzo y el coste final necesario.
- Se tiene el riesgo de no acabar nunca.
- No es recomendable para sistemas en tiempo real, de alto índice de seguridad, de procesamiento distribuido, y/o de altos índices de riesgos.

CICLO DE VIDA ESPIRAL.

El proceso de desarrollo de software se representa como una espiral, donde en cada ciclo se desarrolla una parte del mismo.

Cada ciclo está formado por cuatro:

- **Determinar objetivos**: Se identifican los objetivos, se ven las alternativas para alcanzarlos y las restricciones impuestas a la aplicación (costos, plazos, interfaz…).
- **Análisis de riesgos**: Un riesgo es por ejemplo, requisitos no comprendidos, mal diseño, errores en al implementación, etc. Utiliza la construcción de prototipos como mecanismo de reducción de riesgos.
- **Desarrollar y probar**: Se desarrolla la solución al problema en este ciclo y se verifica que es aceptable.
- **Planificación**: Revisar y evaluar todo lo realizado y con ello decidir si se continúa, entonces hay que planificar las fases del ciclo siguiente.

Sus ventajas son:

- No requiere una definición completa de los requisitos para empezar a funcionar.
- Análisis del riesgo en todas las etapas.
- Reduce riesgos del proyecto.
- Incorpora objetivos de calidad.

Sus inconvenientes son:

- Es difícil evaluar los riesgos
- El costo del proyecto aumenta a medida que la espiral pasa por sucesivas iteraciones.
- El éxito del proyecto depende en gran medida de la fase de análisis de riesgos.

METODOLOGÍAS

Metodología esta compuesta por modos sistemáticos de realizar, gestionar y administrar un proyecto a través de etapas y acciones, partiendo desde la necesidades del producto hasta cumplir con el objetivo para el que fue creado.

Metodologías tradicionales

- RUP

Metodologías ágiles

- XP
- SCRUM
- Kanban

RUP

Dirigido por casos de uso: Los cases de uso guían el proceso de desarrollo ya que los modelos que se obtienen, como resultado de los diferentes flujos de trabajo, representan la realización de los casos de uso (cómo se llevan a cabo).

Centrado en la arquitectura: Se definen los casos de uso principales, los cimientos de la aplicación y sobre ello se va incrementando.

Iterativo e incremental: Una iteración involucra actividades de todos los flujos de trabajo, aunque desarrolla fundamentalmente algunos más que otros

Uso extensivo de documentación.

XP (EXTREME PROGRAMMING)

Se diferencia de las metodologías tradicionales principalmente en que pone más énfasis en la adaptabilidad que en la previsibilidad.

Consideran que los cambios de requisitos sobre la marcha son un aspecto natural, inevitable e incluso deseable del desarrollo de proyectos.

El cliente no sabe lo que quiere hasta que lo tiene.

Esta metodología lleva las buenas prácticas a niveles extremos. Por ejemplo el TDD (test Driven development) fue usado en la NASA en el proyecto mercurio en los 60.

- **Comunicación**: Dentro en un equipo de desarrollo es fundamental para que la información entre los miembros fluya con normalidad, esto ayudará a evitar errores y agilizar las tareas compartidas
- **Retroalimentación (Feedback)**: Con el cliente es de gran ayuda entregar un software de calidad que cumpla las expectativas del mismo cliente. Para ello, el desarrollo debe realizarse en ciclos cortos que permitan mantener una retroalimentación constante y continua.
- **Simplicidad**: Cuando se trabaja en el diseño de una historia de usuario, es importante centrarse en esa tarea exclusivamente para mantener un código limpio y sencillo. Esto ayuda a evitar el over-engineering, es decir, no se debe preparar software "por si acaso".
- **Coraje**: Ser valiente para evitar over.engineering, para desechar un código fuente antiguo inservible, para refactorizar código que funcione o para sobreponerse de problemas que lleva tiempo sin resolverse.

Prácticas:

- Planificación:
  - Historias de usuario
  - Plan de entregas
  - Plan de iteraciones
  - Reuniones diarias
- Diseño:
  - Simplicidad
  - Soluciones
  - Refactorización
  - Metáforas
- Desarrollo de código:
  - Disponibilidad del cliente
  - Uso de estándares
  - Programación dirigidas por pruebas
  - Programación por pares
  - Integración continua
  - Propiedad colectiva del código
  - Ritmo sostenido.
- Pruebas
  - Pruebas unitarias
  - Detección y corrección de errores
  - Pruebas de aceptación

SCRUM

Es un marco de trabajo que define un conjunto de prácticas y roles, y que puede tomarse como punto de partida para definir el proceso de desarrollo que se ejecutará durante un proyecto.

Los roles principales en Scrum son el **Scrum Master**, que procura facilitar la aplicación de scrum y gestionar cambios, el **Product Owner**, que representa a los clientes/usuarios y el **Team developer** que realiza el desarrollo y demás elementos relacionados con él.

Durante cada sprint, un periodo entre una y cuatro semanas (la magnitud es definida por el equipo y debe ser lo más corta posible), el equipo crea un incremento de software potencialmente entregable (utilizable).

El conjunto de características que forma parte de cada sprint viene del *Product Backlog*, que es un conjunto de requisitos de alto nivel priorizados que definen el trabajo a realizar.

Esta metodología se basa:

- El desarrollo incrementa de los requisitos del proyecto en bloques temporales cortos y fijos.
- Se da prioridad a lo que tiene más valor para el cliente.
- El equipo se sincroniza diariamente y se realizan las adaptaciones necesarias.
- Tras cada iteración (un mes o menos entre cada una) se muestran al cliente el resultado real obtenido, para que este tome las decisiones necesarias en relación a lo observado.
- Se le da la autoridad necesaria al equipo para poder cumplir los requisitos.
- Fijar tiempos máximos para lograr objetivos
- Equipos pequeños. (de 3 a 9 personas cada uno).

KANBAN

La palabra Kanban viene del japonés y traducida literalmente quiere decir tarjeta con signos o señal visual. El tablero más básico de Kanban está compuesto por tres columnas: "Por hacer", "En proceso", "Hecho". Si se aplica bien y funciona correctamente, serviría como una fuente de información, ya que demuestra dónde están los cuellos de botella en el proceso y qué es lo que impide que el flujo de trabajo sea continuo e ininterrumpido.

Las tarjetas Kanban visualizan las tareas de trabajo en un flujo de trabajo y le ayudan a:

- Registrar documentación clave de una tarea
- Revisar los detalles de un vistazo
- Minimizar la pérdida de tiempo
- Identificar oportunidades para colaboración.

HERRAMIENTAS DE APOYO AL DESARROLLO DEL SOFTWARE

Las herramientas CASE (Computer Aided Software Engineering) son diversas aplicaciones informáticas destinadas a aumentar la productividad y calidad en el desarrollo de software.

- Objetivos:
    - **Incrementar**:
        - Productividad del equipo
        - Calidad del software
        - Reusabilidad del software
    - **Reducir**:
        - Coste de desarrollo y mantenimiento
    - **Amortizar**:
        - Gestión del proyecto
        - Desarrollo del software
        - Mantenimiento del software.

Clasificación según la fase del ciclo de vida que abordan:

- CASE Frontales (front-end) o **Upper CASE**: Herramienta de apoyo en las primeras fases:
    - Planificación
    - Análisis de requisitos.
- CASE intermedias o **Middle CASE**: Herramienta de apoyo a las últimas fases:
    - Análisis y diseño.
- CASE dorsales (back-end) o **Lower CASE**: Herramientas de apoyo en las últimas fases:
    - Implementación (generación de código)
    - Pruebas
    - Mantenimiento.

Ejemplos de herramientas CASE libres son: ArgoUML, Use Case Maker, ObjectBuilder.

ROLES DEL DESARROLLO:

- **Diseñador del software**: Nace como una evolución del analista y realiza, en función del análisis, el diseño de la solución que hay que desarrollar. Participa en la etapa de diseño.
- **Analista programador**: Comúnmente llamado "desarrollador" domina una visión más amplia de la programación, aporta una visión general del proyecto más detallado, diseñando una solución más amigable para la codificación y participando activamente en ella. Participa en las etapas de diseño y codificación.
- **Programador**: Se encarga de manera exclusiva de crear el resultado del diseño realizado por analistas y diseñadores. Escribe el código fuente del software. Participa en la etapa de codificación.
- **Arquitecto del software**: Es la argamasa que cohesiona el proceso de desarrollo. Conoce e investiga:
    - **Frameworks**: Estructura de soporte definid con módulos de software concreto (soporte de programas, bibliotecas, lenguajes de scripting) que puede servir de base para la organización, desarrollo y unión de diferente componentes del software
    - Tecnologías.
    - Revisa todo el procedimiento para que el desarrollo de software se lleve a cabo de la mejor forma posible y con los recursos más apropiados. Participa en las etapas de análisis, diseño, documentación y explotación.

ENTORNOS DE DESARROLLO (AL FIN ALGO QUE SI TIENE QUE VER CON EL CURSO, PERO NO ENTRARÁ MUCHO EN EL EXAMEN)

DEFINICIÓN DE UN IDE:

"*Integrated Development Enviroment*".

Es un software compuesto por una serie de herramientas que ayudan al desarrollo de software mediante las cuales están enfocadas a dicho fin.

Este conjunto de herramientas puede estar enfocadas a dar soporte a un único lenguaje de programación a varios.

Las principales herramientas que constituyen un IDE son:

- **Editor**: Se utilizan editores que colorean la sintaxis para ayudar al programador a comprender mejor el programa y facilitando la detección de errores.
- **Compilador o intérprete**: Vinculando al lenguaje utilizado, será de un tipo u otro.
- **Depurador**: Permite la ejecución de la aplicación paso a paso para poder inspeccionar, por ejemplo, valores de variables, etc. Un depurador necesita un intérprete.
- **Constructor de interfaz gráfico (GUI)**: Mediante el uso de esta herramienta, el programador, podrá crear componentes gráficos (ventanas, botones, pestañas).

Además de estas funciones, también es frecuente encontrar:

- **Control de versiones**: Permite almacenar los cambios en nuestro código en repositorios remotos y compartirlo con otros programadores.
- **Importación/Exportación de programas**: Permite la portabilidad de nuestro programa entre equipos.

EVOLUCIÓN DE LOS IDE

Los primeros nacen el los años 70, y se popularizan en los 90. Tiene el objetivo de ganar fiabilidad y tiempo en los proyectos de software. Proporcionan una serie de componentes con la misma interfaz gráfica, con la consiguiente comodidad, aumento de eficiencia y reducción de tiempo de codificación.

Normalmente dedicado a un solo lenguaje, aunque con plugins pueden ser compatibles con otros lenguajes.

En la época de la tarjeta perforada el concepto de Entorno de desarrollo Integrado no tenía sentido.

Los programas estaban escritos con diagramas de flujo y entraban al sistema a través de tarjetas perforadas. Posteriormente eran compilados.

El primer lenguaje de programación que usa un IDE fue BASIC con Visual BASIC ahora conocido como Visual Studio.

Este primer IDE estaba basado en consola de comando exclusivamente. Sin embargo, el uso que hace de la gestión de archivos, compilación, depuración… es perfectamente compatible con los IDE actuales.

A nivel popular el primer IDE puede considerarse que fue el IDE llamado Maestro I. Nació en principios de los 70 y fue instalado por unos 22000 programadores en todo el mundo. Lideró durante los 70 y 80.

El uso de los entornos integrados de desarrollo se ratifica y afianza en los 90 y hoy en día contamos con infinidad de IDE, tanto libres como propietario.

FRAMEWORK

Todas las aplicaciones informáticas tienen una estructura que facilita su gestión.

Esta estructura esta constituida por distintos tipos de ficheros: código fuente, ficheros configuración, ficheros binarios, librerías, ejecutables, etc.…

Un **framework** está formado por un conjunto de librerías, herramientas y/o módulos que nos permiten desarrollar aplicaciones en un contexto determinado.

Generalmente no están vinculadas a un lenguaje de programación específico, a mayor nivel de detalle, mayor será la obligatoriedad de vincularse con un lenguaje determinado. Ejemplos:

- **Modelo-Vista-Controlador (MVC)**: Es un sencillo patrón de diseño que nos permite separar las funciones de nuestra aplicación en capas: presentación (vista), datos (modelos) y operaciones (controlador).
- **Laravel**: Es un framework específico para el lenguaje de programación PHP. Este facilita la creación de sitios webs.

PLUGIN

Es un programa informático que mejora o amplía la funcionalidad del programa o aplicación principal.

Los plugin pueden ser desarrollados tanto por empresas como por terceros. La principal diferencia radica en que este último no dispone de certificado de seguridad.

Existen gran variedad de aplicaciones cotidianas o famosas que suelen incluirnos:

- Navegadores web.
- Sistemas gestores de contenido (CMS).
- Reproductores de audio y video.

PRINCIPALES FUNCIONES DE UN ENTORNO DE DESARROLLO.

- Editor de código (colorines).
- Autocompletado de código, atributos y métodos de clases.
- Identificación automática de código.
- Herramientas de concepción visual para crear y manipular componentes visuales.
- Asistentes y utilidades de gestión y generación de código.
- Archivos fuente en unas carpetas y compilados a otras.
- Compilación de proyectos complejos en un solo paso.
- Control de versiones: tener un único almacén de archivos compartido por todos los colaboradores de un proyecto. Ante un error, mecanismo de autorrecuperación a un estado anterior estable.
- Soporta cambios de varios usuarios de maneras simultáneas.
- Generador de documentación integrado.
- Detección de errores de sintaxis en tiempo real.

OTRAS FUNCIONES

- Ofrece refactorización de código: cambios menores en el código que facilitan su legibilidad sin alterar su funcionalidad (por ejemplo cambiar el nombre a una variable).
- Permite introducir automáticamente tabulaciones y espaciados para aumentar la legibilidad.
- Depuración: seguimiento de variables, puntos de ruptura y mensajes de error de intérprete.
- Aumento de funcionalidades a través de la gestión de sus módulos y plugins.
- Administración de las interfaces de usuario (menús y barras de herramientas).
- Administración de las configuraciones del usuario.

CONFIGURACIÓN Y PERSONALIZACIÓN DE ENTORNOS DE DESARROLLO.

Una vez tenemos instalado nuestro entornos de desarrollo podemos acceder a personalizar su configuración.

Al abrir un proyecto existente, o bien crear un proyecto nuevo, seleccionamos un desplegable con el nombre de "configuración" desde el que podremos personalizar distintas opciones del proyecto.

Podemos personalizar la configuración del entorno sólo para el proyecto actual, o bien para todos los proyectos, presentes y futuros.

Parámetros configurables del entorno:

- Carpeta donde se alojarán todos los archivos del proyecto.
- Carpetas de almacenamiento de paquetes y paquetes prueba.
- Administración de la plataforma del entono de desarrollo.
- Opciones de la compilación de los programas: compilar al grabar, generar información de depuración…
- Opciones de empaquetado de la aplicación: nombre del archivo empaquetado (con extensión .jar, que es la extensión característica de este tipo de archivos empaquetados) y momento del empaquetado.
- Opciones de generación de documentación asociada al proyecto.
- Descripción de los proyectos, para una mejor localización de los mismos.
- Opciones globales de formato del editor: número de espaciados en las sangrías, color de errores en la sintaxis, color de etiquetas, opción de autocompletado de código, propuestas de insertar automáticamente código…
- Opciones de combinación de teclas (atajos de teclado).

ACTUALIZACIÓN Y MANTENIMIENTO

Mantener los componentes actualizados

Realizar copias de seguridad sobre las bases de datos de los proyectos para restaurarlos en caso de error.