# UD1.Desenvolvemento de Software

DAM1-Contornos de Desenvolvemento 2024-25


**1.O software** **2**
1.1. Tipos de software 2
1.2. Linguaxes de programación 3
**2. Fases de desenvolvemento dunha aplicación** **7**
Enxeñaría de software 7
2.1. Análise de requisitos 8
2.2. Deseño 11
2.3. Codificación 13
2.4. Probas 15
2.5. Documentación 16
2.6. Explotación 19
2.7. Mantemento 19
**3. Ciclo de vida do software. Modelos** **20**
3.1. Modelo en Cascada 20
3.2. Modelos evolutivos 21
**4. Metodoloxías de desenvolvemento** **24**
4.1. Metodoloxías estruturadas 24
4.2. Metodoloxías orientadas a obxectos 24
4.3. Metodoloxías para sistemas en tempo real 24
4.4. Metodoloxías áxiles 24
**5. Roles no desenvolvemento de software** **26**
5.1. Xefe de proxecto 26
5.2. Analista de software 26
5.3. Arquitecto de software 26
5.4. Desarrollador de software 27
5.5. Programador 27
5.6. Tester 27
**Ferramentas de Deseño e Planificación** **28**
Organigrama (Flowchart) 28
Pseudocódigo (Pseudocode) 29

# 1.O software

O **software** é o conxunto de programas informáticos para executar tarefas nun ordenador xunto cos datos necesarios sobre os que traballa.

Un **programa informático** é un conxunto ordenado de instrucións, escritas nunha linguaxe de programación segundo unhas regras, para realizar unha tarefa en particular dentro dunha computadora. En palabras sinxelas, é unha secuencia de ordes que lle indican a unha computadora que facer.

## 1.1. Tipos de software

1.1.1. Segundo a tarefa que realiza

- **Software de sistema:** é o que fai que o hardware funcione. Está formado por programas que administran a parte física e interactúan entre os usuarios e o hardware. Algúns exemplos son os sistemas operativos, os controladores de dispositivos, as ferramentas de diagnóstico, etc.
- **Software de aplicación:** programas que realizan tarefas específicas para que o computador sexa útil ao usuario. Por exemplo, os programas ofimáticos, o software médico ou o de deseño asistido, etc.
- **Software de desenvolvemento**: proporciona ao programador as ferramentas necesarias para escribir os programas informáticos e para facer uso de distintas linguaxes de programación. Entre eles atopamos os contornos de desenvolvemento integrado (IDE).

1.1.2. Segundo o método de distribución

- **Shareware:** onde os usuarios poden pagar e despois descargar o aplicativo desde internet.
- **Freeware:** onde os usuarios poden descargar o aplicativo de forma gratuíta, pero que mantén os dereitos de autor.
- **Adware:** é un aplicativo onde se ofrece publicidade incrustada, mesmo na instalación do mesmo.

1.1.3. Segundo a licenza de software.

Unha licenza é un contrato entre o desarrollador dun software e o usuario final. Nel especifícanse os dereitos e deberes de ambas as partes. É o desarrollador o que especifica que tipo de licenza distribúe.

- **Software propietario:** pertence exclusivamente a unha persoa ou empresa e o código fonte co que está realizado só pode velo ou modificalo o seu dono.
- **Software libre:** o autor da licenza concede liberdades ao usuario para usar, adaptar compartir ou mellorar.
- **Software de dominio público:** este software non pertence a ningún propietario e carece de licenza, polo que todo o mundo o pode utilizar.

- **Licenza GPL:** Licenza GPL (Licenza Pública Xeral de GNU): A licenza GPL é unha das licenzas de software libre máis populares, creada pola Free Software

Foundation (FSF). Esta licenza garante que calquera persoa poida usar, modificar e distribuír o software, sempre que as modificacións ou distribucións futuras manteñan a mesma licenza. É dicir, calquera versión modificada ou mellorada debe seguir sendo libre e aberta. A GPL asegura que o software permaneza libre para todos os usuarios. Exemplos: núcleo de Linux, WordPress, Gimp, etc.

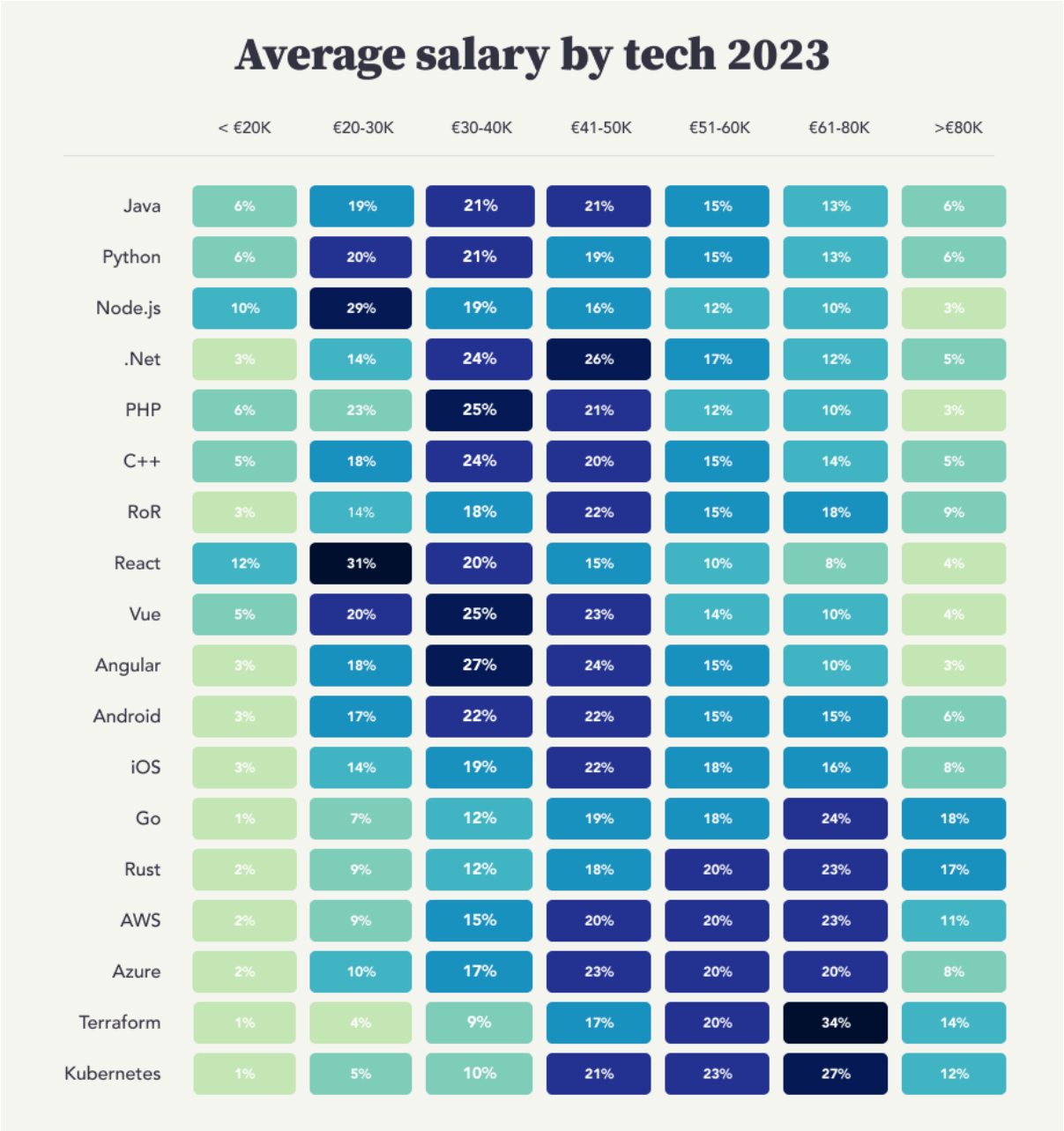
- **Código aberto:** refírese a software cuxo código fonte está dispoñible publicamente, permitindo que calquera persoa poida velo, modificalo e distribuílo. Isto promove a colaboración e o desenvolvemento comunitario, xa que calquera pode contribuír a mellorar o software ou adaptalo ás súas necesidades. O código aberto é unha filosofía que fomenta a transparencia e a innovación. Non todos os software de código aberto teñen que ser libres, pero moitos o son. Exemplos: Mozilla Firefox, Apache HTTP Server, LibreOffice, etc.

## 1.2. Linguaxes de programación

Unha linguaxe de programación pódese definir como unha notación para escribir programas a través dos que se pode establecer unha comunicación co hardware e dar así as instrucións necesarias para realizar unha determinada tarefa. Existen distintas clasificacións das linguaxes de programación, inda que non todas serán obxecto de estudo neste tema.

Comparativas de popularidade, empregabilidade e uso de linguaxes de programación:

- TIOBE Programming Community Index
- PYPL PopularitY of Programming Language
- Top Programming Languages 2022 - IEEE Spectrum
- Stack Overflow Annual Developer Survey




Fuente: [Salarios en tecnología 2023 [España] - Manfred]

**Investiga:** Características principales de los lenguajes más populares.

### 1.2.1. Segundo o nivel de abstracción

O nivel de abstracción, é o modo no que as linguaxes de programación afástanse do código máquina e achéganse cada vez máis á linguaxe que utilizamos hoxe en día.

- **Linguaxe Máquina:** é a linguaxe de programación que entende directamente a máquina (computadora). Esta linguaxe de utiliza o alfabeto binario, é dicir, 0 e 1.
- **Linguaxe de baixo nivel:** Son moito mais fáciles de utilizar que a linguaxe máquina, pero dependen moito da máquina ou computadora como sucedía coa linguaxe máquina. *Ensamblador* foi a primeira linguaxe de programación que tratou de substituír á linguaxe máquina por outra linguaxe que fose máis parecido á dos seres humanos.

- **Linguaxes de alto nivel:** Este tipo de linguaxes de programación son independentes da máquina, podémolos usar en calquera computador con moi poucas modificacións ou sen elas, son moi similares á linguaxe humana, cun alto nivel de abstracción, pero precisan dun programa intérprete ou compilador que traduza esta linguaxe de programación de alto nivel a un de baixo nivel como a linguaxe de máquina que a computadora poida entender.

### 1.2.2. Segundo a forma de execución

- **Linguaxes compiladas:** Ao programar no alto nivel hai que traducir esa linguaxe a linguaxe máquina a través de compiladores. Os compiladores traducen desde un linguaxe fonte a unha linguaxe destino. Devolverá erros se o linguaxe fonte está mal escrito e executarao se a linguaxe destino é executable pola máquina. Exemplo: C, C++, C.
- **Linguaxes interpretadas:** Son outra variante para traducir programas de alto nivel. Cando executamos unha instrución, débese interpretar e traducir ao linguaxe máquina. Exemplos: PHP, JavaScript, Python. A linguaxe Java usa tanto a compilación como a interpretación. Un programa fonte en Java hai que compilalo primeiro a un formato intermedio, chamado bytecodes, para que logo unha máquina virtual o interprete.

**Investiga:** Ventajas y desventajas de lenguajes compilados e interpretados. El caso “especial” de Java.

**Máquina virtual:** é unha aplicación que executa os programas como si fose unha máquina real. Poden ser sistema (emulan un ordenador) ou de proceso (executan un proceso concreto dentro do SO). JVM é a máquina empregada por Java, que permite que un programa previamente compilado poda ser executado en calquera plataforma.

**Linguaxes visuais:** son un tipo de linguaxe que utiliza compoñentes gráficos como iconas, botóns e símbolos en forma de codificación, tamén permite aos usuarios utilizar simplemente unha interface de arrastrar e soltar. Unha vantaxe é que en moitos casos pódese realizar directamente en plataformas web, sen ter que instalar ningún entorno de desenvolvemento. Pola contra, en aplicacións de certa complexidade e nivel pode resultar complexo e difícil de manter.

### 1.2.3. Segundo o seu paradigma de programación.

- **Imperativas:** céntranse en describir como un programa debe lograr un resultado específico. Neste paradigma, o código componse duna serie de instrucións que se executan de xeito secuencial. (Java, Php, Ada, C).
- **Programación Estruturada:** busca escribir código de xeito organizado e fácil de entender mediante a división dun programa en estruturas de control claras e lexibles. Combina o uso da secuencialidade, estruturas de control e a modularidade. (Pascal, Fortran, C).
- **Programación Orientada a Obxectos (POO):** basease no concepto de "obxectos" e céntrase na organización de datos e funcións en unidades chamadas obxectos. (Java, Python, C++, Perl). Algúns conceptos da programación orientada a obxectos son:

    - **Obxectos**: conteñen datos e funcións que operan neses datos. Os obxectos poden representar entidades do mundo real ou conceptos abstractos.
    - **Clases**: son plantillas ou moldes que definen a estrutura e o comportamento dos obxectos. Unha clase describe os atributos (datos) e os métodos (funcións) que os obxectos desa clase terán.
    - **Encapsulación**: é un principio que implica ocultar os detalles internos dun obxecto e expoñer só unha interface pública para interactuar con el.
    - **Herdanza**: permite a creación de novas clases baseadas en clases existentes. Unha clase derivada herda atributos e métodos dunha clase base.
    - **Polimorfismo**: permite que diferentes obxectos respondan de maneira específica ás mesmas chamadas de método. Isto lógrase mediante a implementación de métodos co mesmo nome en diferentes clases.
    - **Abstracción**: é a simplificación de obxectos e procesos complexos en representacións máis simples. Permite modelar obxectos do mundo real de maneira efectiva no software.
    - **Mensaxes**: A comunicación entre obxectos baséase no envío de mensaxes. Os obxectos interactúan chamando a métodos doutros obxectos a través de mensaxes.
    - **Modularidade**: A POO promove a modularidade ao dividir un programa en clases e obxectos, o que facilita a organización e o mantemento do código.
- **Programación Declarativa:** céntrase en describir o que debe facer un programa, en lugar de como debe facelo. (SQL, Prolog).
- **Programación dirixida a eventos:** o fluxo dun programa determínase principalmente por eventos ocorridos no sistema ou na aplicación en lugar dunha secuencia de execución predefinida. Un evento é un suceso ou sinal que ocorre nun sistema (clic nun botón, mover o rato). (Java, Visual Basic).

# 2. Fases de desenvolvemento dunha aplicación

## Enxeñaría de software

É importante destacar que o proceso de desenvolvemento de software non foi concibido nos seus inicios como un proceso de enxeñería, senón como un traballo artesanal no que cada desenvolvedor de software comezaba a programar directamente sen realizar fases previas e sen seguir uns estándares mínimos.

Na década de 1970 produciuse o que se coñece como **crise do software**, que fai referencia a un conxunto de problemas que se atopan no desenvolvemento do software. Esta crise abarcou como desenvolver software, como mantelo e como satisfacer a crecente demanda de software.

Os problemas fundamentais do software foron os seguintes:

- A planificación e a estimación dos custos eran moitas veces moi imprecisas.
- A produtividade dos que desenvolveron o software non se correspondía coa demanda dos seus servizos.
- A calidade do software ás veces nin sequera era a adecuada.

Concluíuse que todos estes problemas podían ser corrixidos e que a clave era **adoptar un enfoque de enxeñería** para o desenvolvemento de software, xunto coa mellora das técnicas e ferramentas.

Todo isto levou ao nacemento da **enxeñaría de software**, unha disciplina que Fitz Bauer definíu como: *o establecemento e uso de principios de enxeñaría para conseguir un software rendible que sexa fiable e funcione de forma eficiente en máquinas reais*.


O obxectivo fundamental da enxeñaría do software é, polo tanto, regular dalgún xeito o desenvolvemento do software, é dicir, establecer con claridade as tarefas que hai que realizar para crear aplicacións informáticas, realizando controis que aseguren que a calidade dos programas resultantes é aceptable.

Para a obtención de software é necesario completar unhas fases nas que se realizan unha serie tarefas:

- **Análise de requisitos:** Especifícanse os requisitos funcionais (como debe comportarse o sistema) e os non funcionais (restricións impostas ao sistema).
- **Deseño:** establece como se vai resolver o problema plantexado.
- **Codificación:** pasar os resultados da fase de deseño a código fonte.
- **Probas:** compróbase si a aplicación funciona correctamente
- **Documentación:** trátase da elaboración de documentos que especifican o funcionamento do programa e serven de soporte tanto para o desarrollador como para o usuario.
- **Explotación:** nesta etapa lévase a cabo a instalación e posta en marcha do software no entorno de traballo do cliente.
- **Mantemento:** realizar as modificacións necesarias para resolver os cambios que se producirán por distintos motivos


## 2.1. Análise de requisitos

O obxectivo desta fase é lograr unha comprensión clara do problema que se precisa resolver ou, é dicir, dos **requisitos funcionais** da aplicación: que funcións terá que realizar a aplicación, que resposta dará a aplicación ás posibles entradas, como se comportará a aplicación en situacións inesperadas, etc.

Esta tarefa non é fácil por varias razóns:

- É posible que o cliente non teña claro que requisitos debe cumprir a aplicación.
- O cliente pode ter claro os requisitos, pero non ser capaz de expresalos dun xeito que sexa comprensible para o equipo de desenvolvemento.
- Pode haber malentendidos entre o equipo de desenvolvemento e o cliente.

Existen outros tipos de requisitos denominados **requisitos non funcionais**, como os seguintes:

- **Fiabilidade**: grao en que unha aplicación funciona sen fallos.
- **Escalabilidade**: capacidade do sistema para manexar aumentos de carga sen diminuír o rendemento.
- **Extensibilidade**: capacidade do software para engadir novas funcionalidades e compoñentes.
- **Seguridade**: o grao en que unha aplicación protexe a información contra o acceso por ela.
- **Mantebilidade**: grao en que se entende, repara ou mellora o software.

Os requisitos non funcionais poden incluir detalles como tempos de resposta do programa, lexislación aplicable, tratamento ante usos simultáneos, etc.

Exemplos de requisitos:

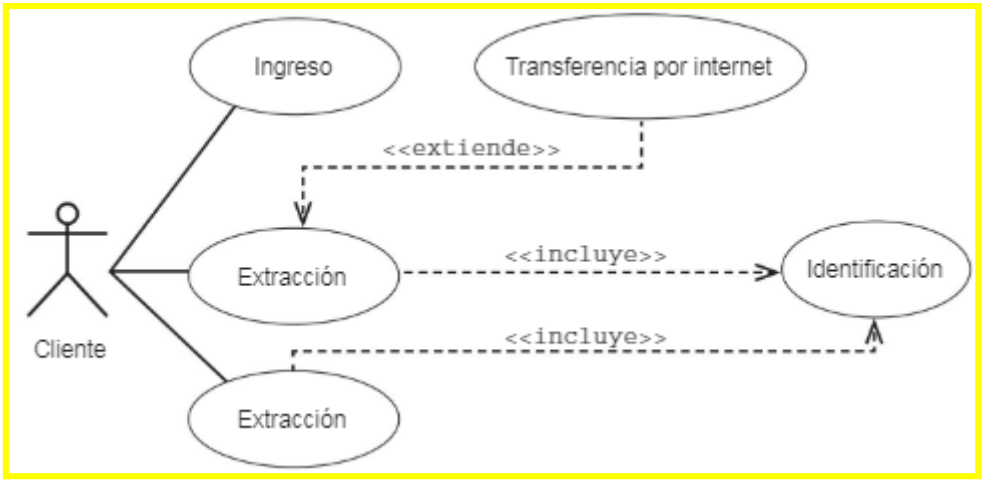
| Requisitos funcionais | Requisitos non funcionais |
|---|---|
| O usuario pode engadir un novo contacto.<br><br>O usuario pode ver unha lista con todos os contactos.<br><br>Da lista de contactos o usuario pode acceder a un contacto.<br><br>O usuario pode eliminar un ou máis contactos da lista.<br><br>O usuario pode modificar os datos de un contacto seleccionado da lista.<br><br>O usuario pode seleccionar certos contactos.<br><br>O usuario pode imprimir a lista de contactos. | A aplicación debe funcionar en sistemas Sistemas operativos Linux e Windows.<br><br>O tempo de resposta a consultas, altas, baixas e modificacións debe ser inferior a 5 segundos.<br><br>Use un sistema de xestión de bases de datos para almacenar os datos.<br><br>Use unha linguaxe multiplataforma para desenvolvemento de aplicacións. interface de usuario é a través de Windows,<br><br>Debe ser intuitivo e fácil de usar.<br><br>A xestión da solicitude realizarase co teclado e rato. Espazo libre no disco, mínimo: 1 GB.<br><br>Cantidade mínima de memoria 2 GB. |

**Amplía** sobre requisitos funcionais e non funcionais:

- Requerimientos Funcionales y No Funcionales, ejemplos y tips | by Requeridos Blog | Medium

Antes de desenvolver os modelos que son froito da fase de análise é necesaria unha tarefa de comunicación co cliente, para a que se poden empregar diferentes técnicas, como:

- **Entrevistas**: é a técnica máis tradicional. O analista entrevista, un por un, aos futuros usuarios do software.
- **Desenvolvemento conxunto de aplicacións (JAD,** *Joint Application Development***)**: créase e reúnese un equipo de analistas e usuarios para traballar xuntos para determinar as necesidades dos usuarios.
- **Planificación Conxunta de Requisitos** (*JRP, Joint Requirements Planning*)
- **Brainstorming**
- **Desenvolvemento dun prototipo**: consiste na construción dun modelo ou maqueta do sistema que permita aos usuarios ver as características que desexan obter. O prototipo pode usarse só para este propósito e descartarse ou mellorarse ata convertirse no produto final.
- **Casos de uso:**
- etc.




Exemplo de diagrama de Casos de Uso que amosa algúns requisitos funcionais dunha aplicación bancaria desde o punto de vista do cliente.

Como resultado desta fase, débese obter un documento denominado **especificación de requisitos de software (ERS)** que serve de base para a seguinte fase de deseño.

No ERS inclúense modelos gráficos con textos de apoio, creados con ferramentas informáticas, e facilmente modificables. Nesta fase, pódense elaborar **diagramas de fluxo de datos** (DFDs), diagramas de fluxo de control (DFC), diagramas de transición de estados (DTE), diagramas Entidade/Relación, diccionario de datos, **diagramas de clases** e **diagramas de casos de uso**, que se estudarán nas UDs 5 e 6.

Outras informacións que pode incluir o ERS son:

- A planificación das reunións que van ter lugar.
- Relación dos obxectivos do usuario cliente e do sistema.
- Relación dos requisitos funcionais e non funcionais do sistema.
- Relación de obxectivos prioritarios e temporización.
- Recoñecemento de requisitos mal expostos ou que conlevan contradicións etc.

Existe unha estrutura para o ERS proposta no estándar 830 do IEEE. Especificación de Requisitos según el estándar de IEEE 830

En parellas/grupos, pensade nunha aplicación sinxela e tratade de analizar os seus requisitos, funcionais e non funcionais, poñendoos por escrito dun xeito ordenado.

Por exemplo:

- Aplicación para compartir coche entre compañeir@s de clase.
- …

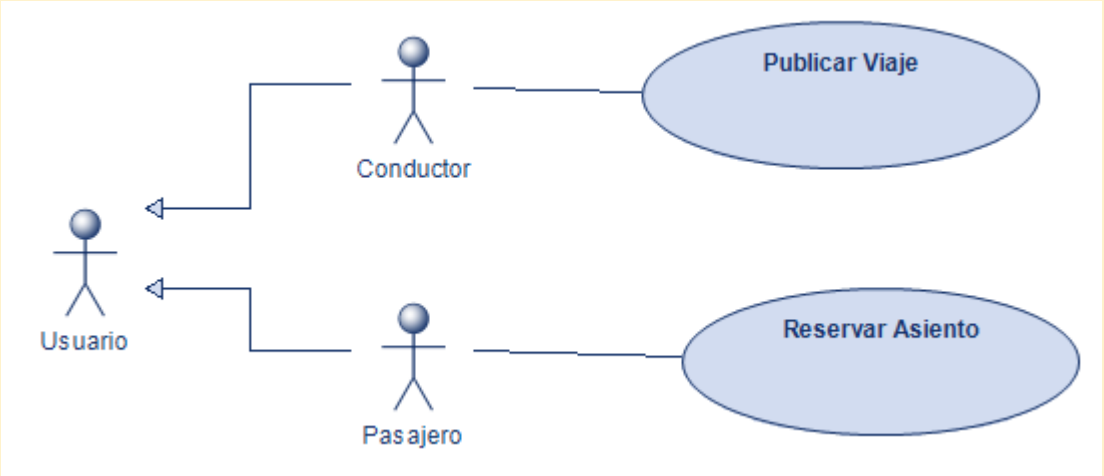
Publicade a idea da aplicación no Foro de Alumnado

Explora ferramentas de deseño de diagramas::

- **Modelio**
- **Umbrello** - Wikipedia, la enciclopedia libre
- PlantUML - Wikipedia, la enciclopedia libre

- draw.io
- Lucidchart
- BOUML - Wikipedia, la enciclopedia libre
- etc.

Crea un diagrama dalgúns Casos de uso da aplicación anterior.




## 2.2. Deseño

Partindo do ERS que describe cal é o problema que se debe resolver, na fase de deseño determínase como se debe resolver dito problema. Para iso partiremos dos diagramas obtidos na fase de análise, perfeccionando algúns deles e creando outros que indiquen os pasos que hai que dar para dar resposta aos requisitos establecidos.

Por exemplo, para cada caso de uso obtido na fase de análise, correspondente en xeral a un requisito funcional específico, xerarase un diagrama de interacción que detalla os pasos necesarios para executalo.

Existen principalmente dous tipos de deseño, o **deseño estruturado**, ou deseño clásico, que se basea no fluxo que seguen os datos a través do sistema; e o **deseño orientado a obxectos** onde o sistema se entende como un conxunto de obxectos que teñen propiedades e comportamentos, que interactúan entre sí, e en eventos que activan operacións que modifican o estado dos obxectos.

O deseño estruturado produce un modelo de deseño con 4 elementos:

- **Deseño de datos.** Encárgase de transformar o modelo de dominio da información creado durante a análise nas *estruturas de datos* que se utilizarán para implementar o software. A disposición dos datos baséase nos datos e relacións definidos no diagrama de relacións entidades ademais dos datos detallados contidos no dicionario de datos.
- **Deseño arquitectónico.** Céntrase na representación da estrutura dos compoñentes do software, as súas propiedades e interaccións. Un compoñente de software pode ser tan sinxelo coma un módulo de programa, pero tamén pode ser tan complicado como unha base de datos ou conectores que permiten a comunicación, a

coordinación e a cooperación entre os compoñentes. Partindo do DFD establécese a estrutura modular do software que se desenvolve.

- **Deseño da interface.** Describe como o software se comunica consigo mesmo, cos sistemas que operan con el e coas persoas que o usan. O resultado desta tarefa é a creación de formatos de pantalla.
- **Deseño a nivel de compoñentes**.Transforma os elementos estruturais da arquitectura do software nunha descrición procedimental dos seus compoñentes. O resultado desta tarefa é o deseño de cada compoñente software cun nivel de detalle suficiente para que poida servir de guía na xeración de código fonte nunha linguaxe de programación. Para realizar este deseño utilízanse representacións gráficas a través de **diagramas de fluxo**, **diagramas de caixas**, **táboas de decisións**, **pseudocódigo**, etc.

**Investiga e proba** distintas ferramentas software para desenvolver distintos tipos de diagramas e pseudocódigo como PSeInt, etc.

- UD1. Pseudocódigo e Diagramas de fluxo con PSeInt

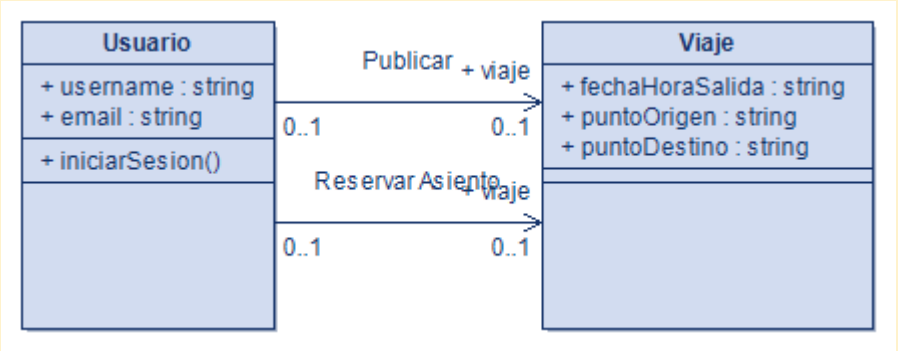
O **deseño de software orientado a obxectos** (OO) é difícil. Para levalo a cabo hai que partir dunha análise orientada a obxectos (AOO). Nesta análise defínense todas as clases que son importantes para o problema a resolver, as operacións e atributos asociados, as relacións e comportamentos e as comunicacións entre clases.

O deseño orientado a obxectos define 4 capas de deseño:

- **Subsistema**. Céntrase no deseño dos subsistemas que implementan as principais funcións do sistema.
- **Clases e Obxectos**. Especifica a arquitectura global de obxectos e a xerarquía de clases necesarias para implementar un sistema.
- **Mensaxes**. Indica como se realiza a colaboración entre obxectos.
- **Responsabilidades**. Identifica as operacións e os atributos que caracterizan a cada clase.

**UML** (*Unified Modeling Language*) utilízase para a análise e deseño orientados a obxectos. É unha linguaxe de modelado baseada en diagramas que se usa para expresar modelos (un modelo é unha representación da realidade onde se ignoran detalles menores). Converteuse no estándar de facto para a maioría das metodoloxías de desenvolvemento orientado a obxectos que existen na actualidade.

Crea un diagrama de clases UML que represente o modelo de datos da aplicación ideada nas actividades anteriores.




## 2.3. Codificación

Proxecto Ejemplo. Aplicación para Compartir Coche
Instalación de XAMPP

Nesta fase tradúcese o deseño a unha linguaxe lexible para o computador. Tamén se farán as probas ou ensaios necesarios para garantir que devandito código funciona correctamente. O resultado desta fase será o **código fonte**.

Ferramentas para a codificación:
- IDEs (vscode, Eclipse, IntelliJ, …)
- Linguaxes de programación (Python, Java, C, JavaScript, PHP, …)
- Control de versións (Git, Subversion, …)
- etc.

En calquera proxecto que se traballe cun grupo de persoas haberá que ter unhas **normas de codificación ou regras de estilo** que sexan sinxelas, claras e homoxéneas. Estas normas facilitarán a corrección e mantemento dos programas e o traballo en equipo.

Estas normas aplícanse a:
- Uso de comentarios
- Formato do código: espaciado, sangrado, liñas en branco, …
- Identificadores e nomes de ficheiros, paquetes, clases, variables, constantes, métodos,
- etc.

**Investiga** en Internet as regras de estilo recomendadas para as linguaxes de programación máis populares.
- Oracle Code Conventions for the Java Programming Language
- Google Java Style Guide
- PEP 8 – Style Guide for Python Code
- Google JavaScript Style Guide
- PSR-1: Basic Coding Standard - PHP-FIG
- PSR-2: Coding Style Guide - PHP-FIG
- etc.

As características desexables de todo código son:

- **Modularidade**: que estea dividido en anacos máis pequenos.
- **Corrección**: que faga o que se lle pide realmente.
- **Lexibilidade**: que sexa fácil de ler: para facilitar o seu desenvolvemento e mantemento futuro.
- **Eficiencia**: que faga un bo uso dos recursos.
- **Portabilidade**: que se poida implementar en calquera equipo. Durante esta fase, o código pasa por diferentes estados:
    - *Código Fonte*: é o escrito polos programadores nalgún IDE ou editor de texto.
    - *Código Obxecto*: é o código binario resultado de compilar o código fonte. Este código non é entendible polo ser humano, pero tampouco pola computadora.
    - *Código Executable*: É o código binario resultante de enlazar os arquivos de código obxecto con certas rutinas e bibliotecas necesarias. Tamén é coñecido como código máquina e xa si é directamente intelixible pola computadora.

## 2.4. Probas

Na etapa de probas xa contamos con software funcional, e trataremos de atopar erros na codificación, na especificación ou no deseño. Durante esta fase realízanse as seguintes tarefas:

- **Verificación**: conxunto de actividades que permiten comprobar se o produto está a construírse correctamente, é dicir, se implementa unha función específica. Entre outras tarefas, revísase o código fonte sen executalo para identificar posibles erros ou vulnerabilidades.
  Por exemplo, se estamos desenvolvendo unha aplicación de cálculo de impostos, a verificación implicaría comprobar se o código que calcula o imposto sobre a renda se
  A verificación responde á pregunta: **"Estamos construíndo o produto correctamente?"**

- **Validación**: conxunto de actividades para comprobar se o produto fai o que ten que facer, é dicir, se o software fai o que se supón que debe facer e se axusta aos requisitos do cliente.
  Continuando co exemplo da aplicación de impostos, a validación implicaría executar o programa con datos reais de contribuíntes e verificar se os cálculos de impostos son correctos e se a interface de usuario é intuitiva e fácil de usar para os clientes.
  A validación responde á pregunta: **"Estamos construíndo o produto correcto?"**

Na etapa de validación planificaranse e executaranse probas que saquen á luz diferentes clases de erros, co menor tempo e esforzo posibles. Unha proba terá éxito se atopamos algún erro non detectado anteriormente.

Un **caso de proba** especifica as condicións previas á proba, os valores de entrada e a saída esperada que debe coincidir coa resultante da execución do software.

As recomendacións para levar a cabo as probas son as seguintes:

- Cada proba definirá os resultados da saída esperados.
- Evitar que o/a programador/a probe os seus propios programas.
- Comprobación de cada resultado en profundidade.
- Incluír todo tipos de datos, tanto válidos e esperados como inválidos e inesperados.
- Comprobar que o software fai o que debe e o que non debe facer.
- Deseñar e documentar probas con coidado.
- Non supoñer que nas probas non se van a cometer erros.
- Cantas máis probas se realicen, maior é a probabilidade de atopar erros e, unha vez solucionados, teremos unha probabilidade maior de poder perfeccionar o noso sistema.

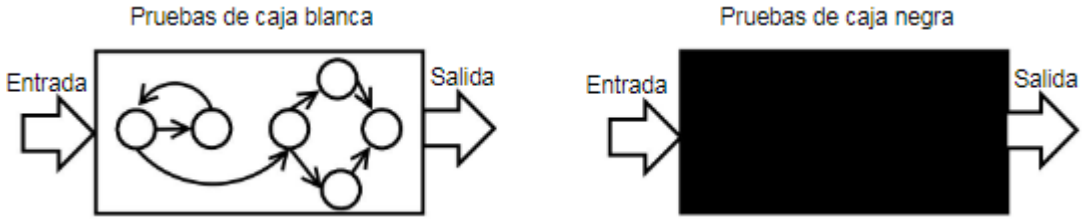
O fluxo do proceso á hora de probar o software é o seguinte:

1. Primeiro xerar un **plan de probas** a partir da documentación do proxecto e da documentación do software para probar.
2. **Deseñar as probas**. Identificar as técnicas a utilizar.
3. **Xerar os casos de proba**.
4. Definir os procedementos de proba. Como, cando e quen realizará as probas.
5. **Executar das probas** aplicando os casos de proba.

6. Avaliar. Identificar de posibles erros. Realizar un informe indicando que casos de proba pasaron, cales non e que fallos se detectaron.
7. **Depurar**. Localizar e corrixir erros. Se se corrixe un erro, voltar a probar o software para verificar que se resolveu o problema. Se non se consegue atopar o erro pode ser necesario realizar máis probas para obter máis información.
8. Analizar os erros para predecir a fiabilidade do software e mellorar os procesos de desenvolvemento.

Para levar a cabo o deseño de casos de proba, existen diferentes tipos de probas:

- **Probas de caixa branca** validan a estrutura interna do sistema. Requiren coñecer o funcionamento interno do programa.
- **Probas de caixa negra** validan os requisitos funcionais sen observar o funcionamento interno do programa.




- **Probas unitarias**, consiste en probar, unha a unha, as diferentes partes do software e comprobar o seu funcionamento (por separado, de maneira independente).
- **Probas de integración**, realízanse unha vez que se realizaron con éxito as probas unitarias, e consistirán en comprobar o funcionamento do sistema competo (con todas as súas partes interrelacionadas).
- **Beta Test ou Proba final**, realízase sobre a contorna de produción onde será usado polo cliente.

Ferramentas de Probas:

- **JUnit**
- PyTest
- PHPUnit
- Selenium
- Apache JMeter
- …

## 2.5. Documentación

Cada etapa do desenvolvemento ten que quedar perfectamente documentada.

Os distintos documentos xerados:

- Poden clasificarse segundo o destinatario e o nivel técnico das súas descricións.
- Facilitan a comunicación entre os membros do equipo de desenvolvemento.
- Inclúen información do sistema que será utilizada polo persoal de mantemento.
- Deben especificar ao usuario como debe usar e administrar o sistema.

A documentación pode dividirse en dúas clases:

- **Documentación do proceso.** Rexístrase o proceso de desenvolvemento e mantemento. Inclúense plans, estimacións e horarios que se usan para predicir e controlar o proceso de software.
- **Documentación do produto.** Describe o produto que está a ser desenvolvido e inclúe a *documentación do sistema* e a *documentación do usuario*. Utilízase sobre todo despois de que o sistema entre en funcionamento.

### 2.5.1. Documentación do usuario

Cada tipo de usuario, recibirá un tipo de documentación diferente. Distinguimos entre os usuarios finais e os adminsitradores do sistema.

| Documento | Destinatario | Contido |
|---|---|---|
| Descrición funcional do sistema | Avaliadores do sistema | Describe funciones xerais. Serve de manual introductorio que permite ao usuario decidir si o sistema é o que necesita. |
| Documento de instalación do sistema | Administradores | Describe como instalar o sistema. Contén información dos ficheiros do sistema, a súa configuración, o hardware mínimo requirido, como iniciar o sistema, etc. |
| Manual introdutorio | Usuarios noveles | Explicacións sinxelas de como empezar a usar o sistema.Inclúe exemplos de uso e solución a erros de usuarios principiantes. |
| Manual de referencia do sistema | Usuarios experimentados | Descrición detallada do sistema e lista completa de erros e como recuperarse deles. Debe ser completo, formal e descriptivo. |
| Guía do administrador do sistema | Administradores | Para sistemas nos que hai comandos de control, descríbense as tarefas do operador, as mensaxes producidas e as respostas do operador. |
| Tarxeta de referencia rápida | | Serve para realizar procuras rápidas de información que evite aos usuarios consultar manuales máis densos. |

### 2.5.2. Documentación do sistema

Describen o sistema, especificacións de requisitos, probas de aceptación, etc. Incluirán:

| Documento | Contido |
|---|---|
| Fundamentos do sistema | Describe os obxectivos do sistema. |
| Análise e especificación de requisitos | Información exacta dos requisitos acordados entre as partes implicadas: usuarios, clientes, desenvolvedores. |

| | |
|---|---|
| Deseño | Describe a arquitectura do sistema, como se descompón o sistema en unidades, a función de cada unha, como interactúan, etc. |
| Implementación | Descrición do sistema nunha linguaxe de programación formal. Comentarios e documentación do código. |
| Plan de probas do sistema | Descrición da avaliación individual das unidades do sistema e as probas de integración de todo o sistema. |
| Dicionarios de datos | Descricións dos termos que se relacionan co software en cuestión. |

En sistemas pequenos, a documentación pode ser menos amplia, pero debería incluir como mínimo a especificación de requisitos, o deseño arquitectónico e o código fonte.

Por desgracia, a miúdo descoidase o mantemento da documentación, xerando problemas entre usuarios e mantedores do sistema, e quedando o usuario desprotexido ante as dúbidas e os erros da aplicación.

**2.5.3. Estrutura do documento**

O documento debe ter unha organización para que, á hora de consultalo, se localice facilmente a información, polo que terá que estar dividido en capítulos, seccións e subseccións. Algunhas pautas son as seguintes:

- Deben ter unha portada, tipo de documento, autor, data de creación, versión, revisores, destinatarios do documento e a clase de confidencialidade deste.
- Debe posuír un índice con capítulos, seccións e subseccións.
- Incluír ao final un glosario de termos

## 2.6. Explotación

Nesta etapa se leva a cabo a instalación e posta en marcha do produto no contorno de traballo do cliente, ou *contorno de produción*.

Nesta etapa realizaranse as seguintes tarefas:

- **Definir a estratexia para a implantación.** Desenvólvese un plan que definen os procedementos para recibir, rexistrar, solucionar, facer seguimento dos problemas e probar o produto no contorno de produción.
- **Probas de operación.** Para cada lanzamento (ou *release*) do produto levaranse a cabo probas que deberán ser satisfactorias antes de liberar o software para uso operativo.
- **Uso operacional do sistema.** O sistema entrará en acción no contorna previsto dacordo coa documentación de usuario.
- **Soporte ao usuario.** Deberase proporcionar asistencia e consultoría aos usuarios cando o soliciten. As peticións e accións que se fagan deberán rexistrarse e supervisarse.

## 2.7. Mantemento

O mantemento do software defínese, segundo a norma IEEE 1219, como a modificación dun produto software despois da entrega para corrixir fallos, mellorar o rendemento ou outros atributos ou para adaptar o produto a un contorno modificado.
Existen catro tipos de mantemento que dependen das demandas do usuario:

1. **Mantemento adaptativo.** Co paso do tempo, pódense producir cambios no entorno orixinal (CPU, sistema operativo, regras da empresa, etc.) para o que se desenvolveu o software. O mantemento adaptativo ten como obxectivo modificar o produto para que se *adapte* a estes cambios. Este tipo de mantemento é o máis habitual debido aos rápidos cambios que se producen na tecnoloxía informática, que na maioría dos casos deixan obsoletos os produtos software desenvolvidos.
2. **Mantemento correctivo.** É moi probable que despois da entrega do produto, o cliente atope erros ou defectos, a pesar das probas e verificacións realizadas nas fases anteriores. Este tipo de mantemento ten como obxectivo *corrixir* avarías descubertas.
3. **Mantemento perfectivo.** A medida que o cliente utiliza o software, pode descubrir funcións adicionais que che poderían aportar beneficios. O mantemento perfectivo é a modificación do produto software destinada a *incorporar novas funcionalidades* (máis aló dos requisitos funcionais orixinais) e novas melloras sobre o rendemento ou a capacidade de mantemento do produto.
4. **Mantemento preventivo.** Consiste en modificar o produto software sen alterar as súas especificacións, co fin de mellorar e facilitar as tarefas de mantemento. Este tipo de mantemento realiza cambios nos programas para que se poidan corrixir, adaptar e mellorar con maior facilidade, por exemplo, os programas poden ser reestruturados para *mellorar a súa lexibilidade* ou engadir novos comentarios que faciliten a súa comprensión. Este mantemento tamén se denomina *reenxeñería de software*.

# 3. Ciclo de vida do software. Modelos

- Modelos de ciclo de vida: Desarrollo en cascada
- Modelos de ciclo de vida: Desarrollo incremental
- Modelos de ciclo de vida: Desarrollo evolutivo

O **ciclo de vida do software** “é a descrición das distintas formas de desenvolver un proxecto ou aplicación informática”.

Permítenos identificar, administrar e planificar a xestión de recursos ata alcanzar un obxectivo proposto. A cantidade de fases que atopamos en cada proxecto variará segundo as necesidades de cada empresa.

Os modelos permítennos establecer unha metodoloxía que nos servirá de guía para validar todas as partes do noso proxecto. A continuación, imos ver cales son os distintos modelos que se utilizan no desenvolvemento e a preparación dun software.

## 3.1. Modelo en Cascada

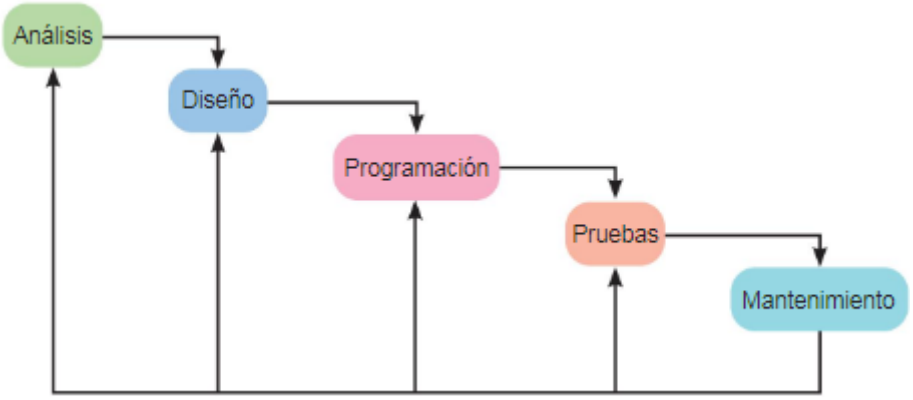
O modelo en cascada, ou **ciclo de vida clásico**, é un método de xestión que divide un proxecto en distintas fases secuenciais. Todas elas funcionan de forma lineal, é dicir, que cada parte do proxecto complétase antes de empezar coa seguinte.


Este modelo é simple e fácil de usar, xa que todo está ben organizado e as fases non poden mesturarse entre si. Sen embargo, presuponse que non haberá cambios nin erros no software, polo que é case imposible de utilizar nun contorno real.

### 3.1.1. Modelo en cascada con retroalimentación

No modelo en cascada faise unha revisión ao completar cada fase e antes de pasar á seguinte. Esta revisión realízase fundamentalmente sobre a documentación producida nesa fase e faise dun xeito formal.

Se durante a realización dunha fase se detectan erros en fases anteriores, será necesario refacer parte do traballo voltando a un punto anterior do ciclo de vida. Por exemplo, se na fase de mantemento de detecta que se cometeu un erro na fase de deseño, será necesario corrixir dito erro de deseño e realizar cambios nas fases posteriores de programación e probas. Esta variante do modelo en cascada denomínase **modelo en cascada con retroalimentación**.




Entre as súas vantaxes podemos destacas as seguintes:

- Fácil de comprender, de planificar e de seguir.
- Existen ferramentas que soportan este modelo.
- É util cando hai unha visión clara do produto final e non hai requisitos ambigüos nin cambiantes.

Algúns inconvenientes do modelo en cascada son:

- Todos os requisitos deben estar definidos desde o principio, o que, a miúdo, é difícil para os clientes.
- Se hai erros é difícil volver atrás nas etapas do desenvolvemento do software.
- O cliente debe ter paciencia xa que non haberá unha versión funcional do software ata que o proxecto estea moi avanzado.

- las 10 mejores herramientas de gestión de proyectos en cascada en 2024

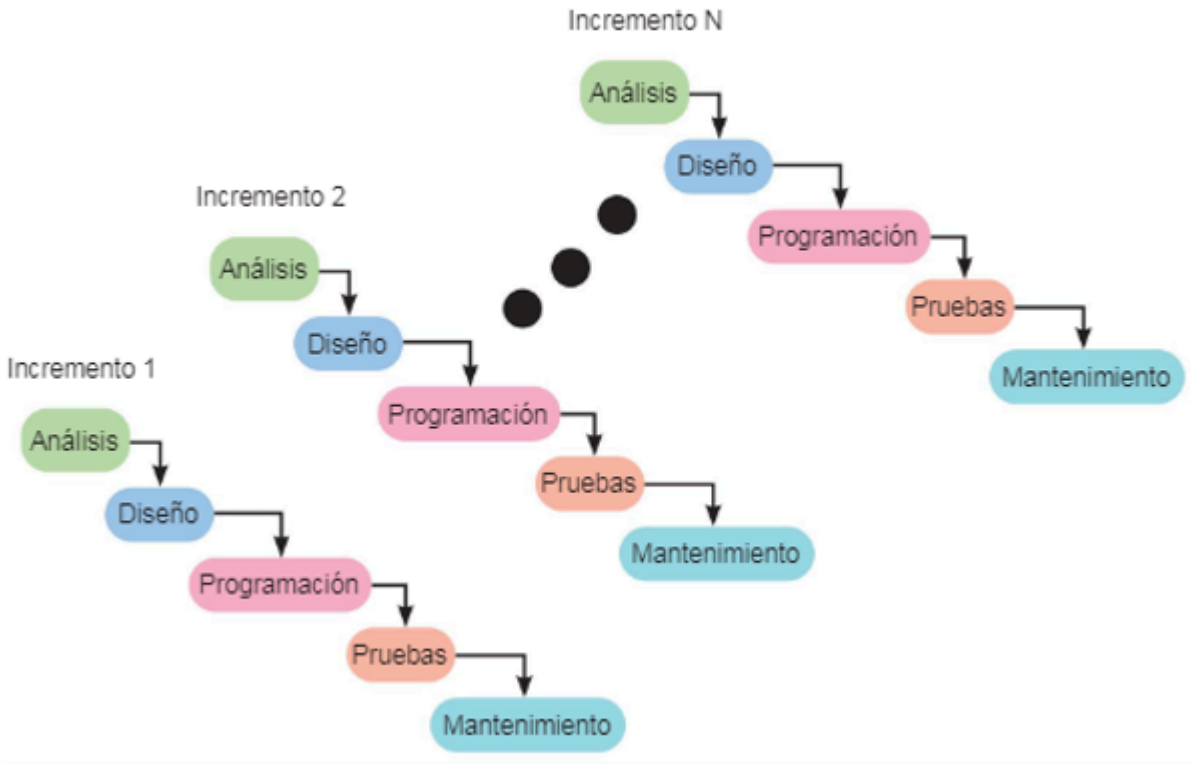
Hoxe en día, o desenvolvemento de software está sometido a multitude de cambios, polo que este modelo de ciclo de vida non resulta moi apropiado na maioría dos casos.

## 3.2. Modelos evolutivos

Teñen en conta a natureza cambiante e evolutiva do software

### 3.2.1. Modelo iterativo incremental.

Tamén se coñece como Modelo baseado en prototipos. Está baseado en varios ciclos en cascada con retroalimentación. Empregando este modelo, podemos entregar o software en partes pequenas pero utilizables.




As vantaxes deste modelo son as seguintes:

- Non se necesita coñecer todos os requisitos.
- Permite unha entrega temperá do programa ao cliente final polo que se facilita a retroalimentación.

Neste modelo tamén atopamos algúns inconvenientes.

- Difícil estimación do esforzo e do custo do programa.
- Poida que non se acabe nunca.
- Non recomendable para desenvolvemento de sistemas en tempo real.

As ferramentas máis coñecidas que utilizan este tipo de modelo de xestión de proxectos de software son Scrum e Kanban, tamén chamadas **Metodoloxías Áxiles**.

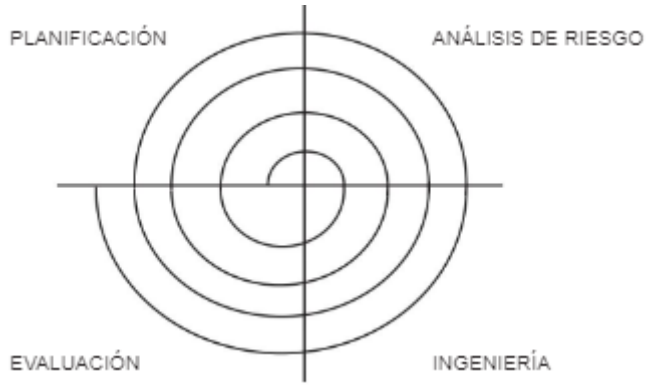
**3.2.2. Modelo en espiral.**

O desenvolvemento de software con este modelo represéntase como unha espiral onde cada ciclo desenvolve unha parte do programa. Entre as súas vantaxes podemos atopar as seguintes:

- Non se necesita unha definición completa dos obxectivos para empezar a funcionar.
- Analízase o risco en todas as etapas.
- Reduce os riscos do proxecto.
- Incorpora obxectivos de calidade.

Doutra banda tamén atopamos algúns inconvenientes.

- Avaliación difícil dos riscos.
- Cantas máis iteracións hai no proxecto, máis elevado é o custo.
- O éxito do proxecto depende da análise de riscos.




Cada ciclo está formado por catro fases.

- **Planificación.** Determinación de obxectivos. Comeza identificando os obxectivos, as alternativas para alcanzalos e as restricións impostas á aplicación das alternativas.
- **Análise de riscos.** Avalíanse as alternativas en relación cos obxectivos e limitacións. Con frecuencia, neste proceso identifícanse os riscos involucrados e a maneira de resolvelos. Utiliza a construción de prototipos como mecanismo de redución de riscos
- **Enxeñaría.** Desenvolver e probar o software. Desenvólvese unha solución e verifícase se é aceptable.
- **Avaliación.** Revísase e avalíase todo o que se fixo. Despois, decídese se se continúa. Se se continúa, débense planificar as fases do ciclo seguinte.

# 4. Metodoloxías de desenvolvemento

Unha metodoloxía de desenvolvemento define un estilo ou unha forma de guiar o proceso de desenvolvemento dun proxecto informático.

## 4.1. Metodoloxías estruturadas

Pásase dunha visión máis xeral do problema con un nivel de abstracción alto a un nivel de abstracción máis baixo.

*Nivel de abstracción: xeneralización dun modelo ou algoritmo, totalmente independente de calquera implementación específica.

## 4.2. Metodoloxías orientadas a obxectos

As metodoloxías orientadas a obxectos son enfoques de desenvolvemento de software que se basean no paradigma da programación orientada a obxectos (POO). Estas metodoloxías céntranse no deseño e desenvolvemento de software utilizando conceptos como obxectos, clases, herdanza, encapsulación e polimorfismo.

## 4.3. Metodoloxías para sistemas en tempo real

As metodoloxías para sistemas en tempo real utilízanse no desenvolvemento de sistemas e aplicacións que deben responder a eventos ou estímulos dentro de límites de tempo específicos e predicibles. Estes sistemas son críticos en áreas como a automoción, a aviación, a industria manufactureira e o control de procesos.

## 4.4. Metodoloxías áxiles

As metodoloxías áxiles son métodos de xestión que permiten adaptar a forma de traballo ao contexto e natureza dun proxecto, baseándose na flexibilidade e a inmediatez, e tendo en conta as esixencias do mercado e dos clientes. Os pilares fundamentais das metodoloxías áxiles son o traballo colaborativo e en equipo.

- Aforrar tanto en tempo como en custos (son baratas e máis rápidas).
- Mellora a satisfacción do cliente.
- Mellora a motivación e implicación do equipo de desenvolvemento.
- Mellora a calidade do produto.
- Eliminar aquelas características innecesarias do produto.
- Alerta rapidamente tanto de erros como de problemas.

#### 4.4.1.Scrum

Dende as necesidades do cliente, construción do produto de forma incremental a través de iteracións. Estas iteracións (en Scrum chámanse Sprint) repítense de forma continua ata que o cliente dá por pechada a evolución do produto.

Características específicas de SCRUM:

- Ciclo de vida iterativo ou incremental, que é aquel en que se vai liberando o produto por partes, cada entrega é o incremento de funcionalidade respecto á anterior. Cada período de entrega → Sprint.
- Reunión diaria. Máximo 15 minutos. Que se fixo o día anterior, que se vai a facer hoxe e que problemas atopáronse.
- Reunión de revisións do Sprint. Ao final de cada sprint, fálase do que se completou e do que non.
- Retrospectiva do Sprint. Tamén ao final de Sprint, serve para que os implicados dean as súas impresións sobre o Sprint; utilízase para a mellora do proceso.

#### 4.4.2. Programación extrema

Metodoloxía áxil centrada en potenciar as relacións interpersoais como clave para o éxito en desenvolvemento do software, promovendo o traballo en equipo, preocupándose pola aprendizaxe dos desarrolladores e propiciando un bo clima de traballo.

Baséase en retroalimentación continua entre o cliente e o equipo de desenvolvemento. É especialmente adecuada para proxectos con requisitos imprecisos e moi cambiantes.

Características específicas da Programación extrema:

- Valórase ao individuo e as interaccións do equipo de desenvolvemento sobre o proceso e as ferramentas. A xente é o principal factor de éxito dun proxecto software.
- Desenvolver un software que funcione, máis que conseguir unha boa documentación.
- A colaboración co cliente. Proponse que exista unha interacción constante entre o cliente e o equipo de desenvolvemento.
- Responder os cambios. A habilidade de responder os cambios que poidan xurdir ao longo do proxecto determina tamén o éxito ou o fracaso de leste. A planificación non debe ser estrita, senón flexible e aberta.

#### 4.4.3. Kanban

Kanban é unha palabra xaponesa que significa tarxetas visuais. Características específicas de Kanban:

- Ten varias barras de progreso onde a tarefa vai avanzando segundo o seu estado.
- Ter unha visión global de todas as tarefas pendentes, das tarefas que estamos a realizar e das tarefas que realizamos.
- Pódense facer cálculos de tempo coas tarefas finalizadas e tarefas similares que imos realizar, para ter unha idea do tempo cada un pode tardar a realizalas.

# 5. Roles no desenvolvemento de software

Un equipo de desenvolvemento de software está formado por moitas persoas con funcións diferentes e polo tanto, con habilidades distintas. E é precisamente a aportación desas capacidades as que levan ao cumprimento dos obxectivos.

## 5.1. Xefe de proxecto

É o **máximo responsable** de que o proxecto saia adiante. Leva a cabo a planificación e posta en marcha do proxecto, a súa execución, seguimento, control e peche.

É a persoa que xestiona o bo funcionamento do proxecto, quen controla e administra os recursos (tanto persoais como económicos) co fin de cumprir o plan e o obxectivo definido. Encárganse de que todo funcione segundo o establecido, de resolver desviacións no plan, e de facer que os diferentes equipos do proxecto se sincronicen e traballen xuntos (distribución de tarefas, fluxo de actividades, tarefas administrativas, contrato co cliente, dirección e control). Ademais, é a cara visible fronte ao cliente, a quen lle informa dos avances e o estado do proxecto. A súa misión é cumprir coas expectativas do cliente.

Ten que ter tanto o coñecemento técnico (coñecer a tecnoloxía e os recursos cos que vai traballar), como a habilidade de xestionar persoas e outros recursos. Ten que ser capaz de traducir o proxecto nun proceso, prevendo desviacións e posibles camiños ata chegar ao obxectivo.

## 5.2. Analista de software

Intervén nas primeiras fases do proxecto onde se realizan as especificacións das necesidades ou a problemática do cliente, desde o xeral ao detalle.

Como experto no problema do cliente, o analista de software traballa xunto a este para definir as especificacións técnicas do produto correctamente. Ademais, ten a misión de traducir eses problemas do cliente en especificacións con sentido para o resto do equipo que logo vai desenvolver o produto.

O analista de software deberá ter unha boa capacidade de comunicación para saber traducir os requirimentos do cliente a instrucións para o equipo, e ademais, traballar de xeito conxunto co cliente.

Este rol tamén se coñece como **analista funcional**.

## 5.3. Arquitecto de software

A partir do traballo do analista debe definir as liñas maestras do deseño, establecendo a arquitectura do sistema, é dicir, os compoñentes técnicos no que se dicidirá e a relación entre eles.

É a persoa ou persoas co suficiente coñecemento técnico do produto ou servizo como para buscar a súa aplicación técnica ás necesidades do cliente. Ten como misión crear, durante todo o proceso de desenvolvemento, a documentación que recolle os requisitos (xunto co analista de software), e será el quen centralice as decisións técnicas sobre os problemas que irán xurdindo, asegurar a calidade, e mellorar continuamente a arquitectura.

Un arquitecto de software terá en conta tanto os requisitos técnicos e funcións, como os requisitos non funcionais. Definiraos xunto ao analista e os priorizará.

Terá que seleccionar a tecnoloxía que se vai a empregar, tendo en conta diversos factores como o custo, as licenzas, a relación cos provedores, a estratexia, a política de actualización.

Sendo este rol uno dos máis importantes e menos coñecidos, o arquitecto de software debe ser un perfil con dotes de facilitador, formador e líder.

## 5.4. Desarrollador de software

Será quen reciba a documentación creada polo arquitecto e o analista, e quen implemente o produto segundo esta.

Este perfil coñece e é capaz de realizar todas as tarefas de desenvolvemento, pero cínguese á implementación e delega outras funcións (como a de programación, o testeo, a supervisión ou o mantemento) a outros membros do equipo. A súa responsabilidade é máis ampla, e ten como misión que todos os aspectos da implementación do proxecto funcionen ben.

A súa diferenza cos analistas pode ser moi sutil, xa que o desarrollador pode participar na definición do produto, nas especificacións e requirimentos, no deseño e mellora de prototipos, ou mesmo a análise do custo e beneficios de elixir un tipo de arquitectura ou outra.

## 5.5. Programador

É o encargado de traducir en código a especificación do sistema. A pesar de que o desarrollador tamén pode “picar código”, os programadores dedícanse exclusivamente a isto. Esta persoa debe coñecer as diferentes linguaxes de programación. E ademais, encárgase de depurar os erros, implementar novas funcionalidades ou manter de forma xeral as aplicacións cando o necesiten. Isto non quere dicir que un programador non poida coñecer de orzamentos, planeación ou requirimentos. Dependerá da experiencia.

## 5.6. Tester

Encargarase de asegurar que os requisitos definidos polo arquitecto de software cúmprense na implementación do produto ou servizo realizada polos desarrolladores e/ou programadores. Para iso, será responsable de aplicar diferentes métodos de testeo xunto aos programadores. Informará de todos os erros atopados durante a fase de probas.

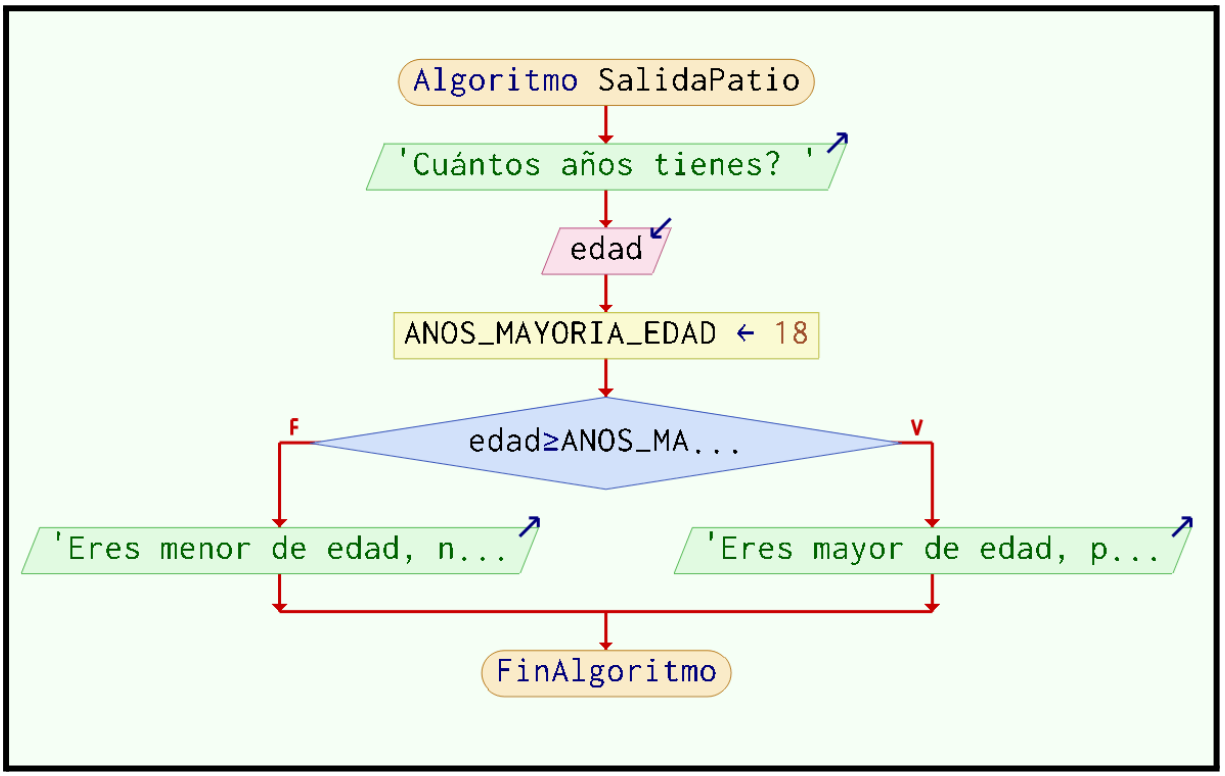
# Ferramentas de Deseño e Planificación

Os organigramas e o pseudocódigo son dúas ferramentas utilizadas no deseño e a planificación de programas informáticos. Ambas axudan a visualizar e comunicar a lóxica e o fluxo dun algoritmo antes da súa implementación nunha linguaxe de programación específico.

## Organigrama (*Flowchart*)

Un organigrama é unha representación gráfica dun algoritmo ou proceso. Utiliza símbolos e frechas para representar as instrucións e o fluxo de control nun programa. Aquí están algúns dos elementos crave dun organigrama:

- **Inicio/Fin:** Represéntase cun óvalo e marca o inicio e o final do proceso.
- **Entrada/Saída:** Representase cun paralelogramo, significa a entrada ou saída de datos.
- **Proceso:** Represéntase cun rectángulo e contén unha acción ou unha serie de accións que se deben realizar.
- **Decisión:** Represéntase cun rombo e utilízase para tomar decisións (xeralmente con respostas "Si" ou "Non").
- **Conexións:** Utilízanse frechas para conectar os símbolos e mostrar a secuencia de execución.
- **Saída por consola:** Utilízanse para mostrar datos por pantalla.




Os organigramas son útiles para visualizar o fluxo dun programa, identificar posibles problemas lóxicos e comunicar a lóxica do algoritmo a outros.

# Pseudocódigo (*Pseudocode*)

O pseudocódigo é unha forma de escribir algoritmos utilizando unha linguaxe de programación simplificado e non estándar. O obxectivo do pseudocódigo é describir a lóxica do programa nun formato que sexa fácil de entender antes de traducilo a unha linguaxe de programación real. Algúns aspectos clave do pseudocódigo son:

- **Sintaxe simple:** O pseudocódigo utiliza unha sintaxe simplificada que se asemella a unha linguaxe de programación real pero sen regras estritas.
- **Lexibilidade:** O pseudocódigo enfócase en ser lexible para humanos e na descrición da lóxica do algoritmo.
- **Uso de palabras chave:** Pode utilizar palabras chave como "Se", "Entón", "Para", "Mentres", etc., para expresar estruturas de control.
- **Non é executable:** O pseudocódigo non se pode executar directamente nunha computadora, pero serve como unha ferramenta de deseño e documentación.

```
Algoritmo SalidaPatio
        Escribir "Cantos anos tes? "
        Leer idade

        ANOS_MAIORIA_EDAD <- 18

        Si idade >= ANOS_MAIORIA_EDAD Entonces
                Escribir "Es maior de idade, podes sair do centro"
        SiNo
                Escribir "Es menor de idade, necesitas unha autorización para sair do centro"
        Fin Si
FinAlgoritmo
```

O pseudocódigo permite que os programadores deseñen e comuniquen a lóxica dun programa antes de escribir o código real nunha linguaxe de programación específico.