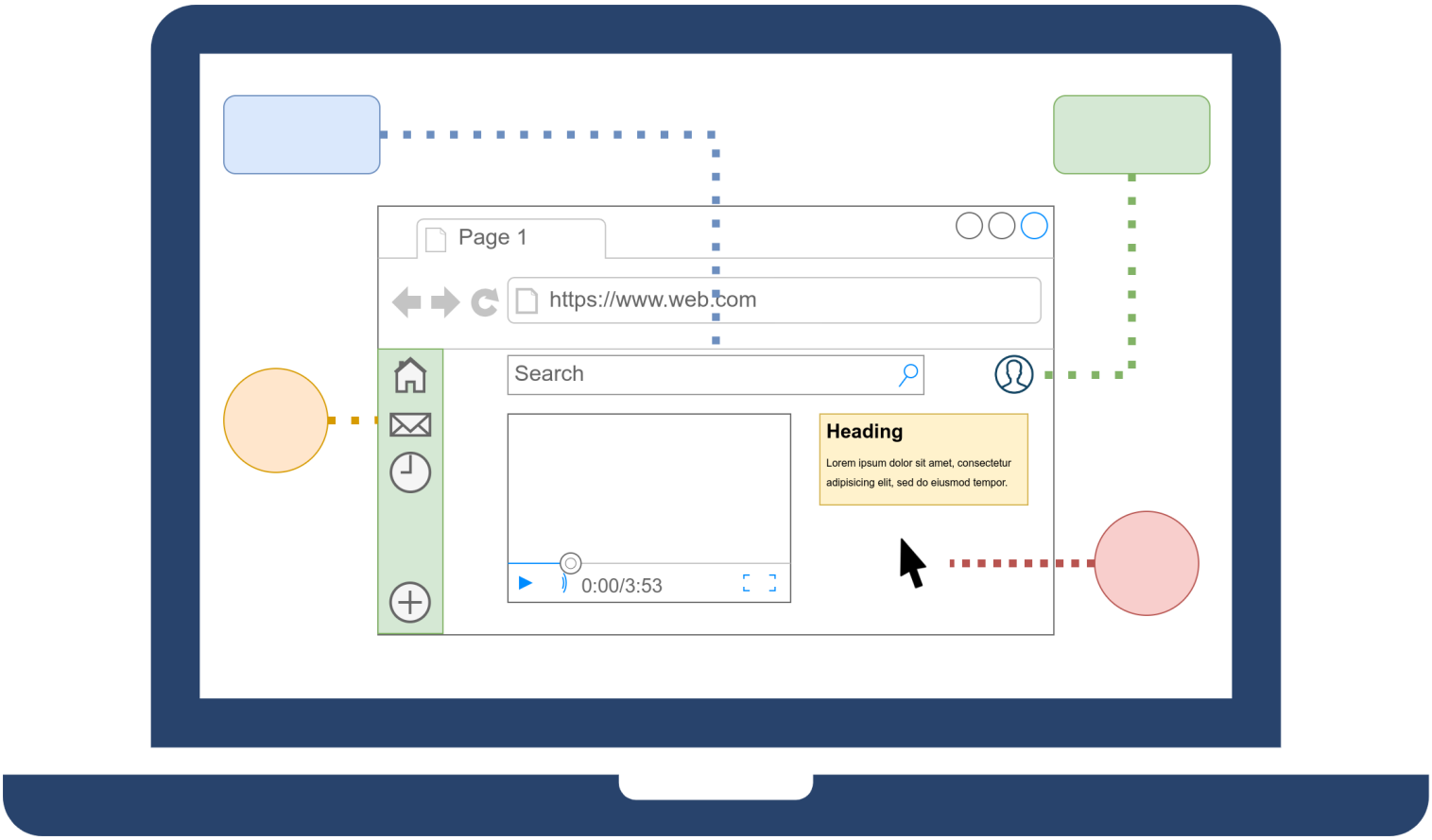
Técnico superior en Desarrollo de Aplicaciones Multiplataforma



Desarrollo de Interfaces

Rubén Gómez Olivencia

2024-2025

- Github: https://github.com/yuki

Puedes encontrar la última versión de este libro en formato **HTML** en el siguiente link, así como otros libros que he creado. Para descargar el código fuente en formato **Markdown** visita el repositorio en GitHub.

Información

Por favor, ponte en contacto conmigo si encuentras algún fallo, falta de ortografía o quieres mejorar de alguna manera este libro. Gracias.

I Historia de las interfaces

II Android UI

III Desarrollo de aplicaciones

Historia de las interfaces

1. Introducción

Una **interfaz** (en plural: interfaces) se utiliza en informática para nombrar a la conexión funcional entre dos sistemas, programas, dispositivos o componentes de cualquier tipo, que proporciona una comunicación de distintos niveles, permitiendo el intercambio de información.

Ejemplos:

- **Interfaces de usuario**: a la hora de hacer uso de un programa.
- **Dispositivo de interfaz humana**: de las siglas en inglés **HID** (*human interface device*), cuando hacemos referencia al interfaz *hardware* que utilizamos los humanos para interactuar con el ordenador.
- **Interfaz física**: un componente que se conecta a otro (puerto USB o puerto SATA).

2. Evolución de las interfaces de usuario

La **interfaz de usuario**, IU (del inglés *User Interface*, **UI**), es el medio que permite la comunicación entre un usuario y una máquina. Normalmente suelen ser fáciles de entender y fáciles de utilizar, aunque en el ámbito de la informática es preferible referirse a que suelen ser "usables, amigables e intuitivas".

Hoy en día podemos diferenciar entre 3 tipos:

- **Interfaz de línea de comandos** (Command-Line Interface, **CLI**): Interfaces alfanuméricas (intérpretes de comandos) que solo presentan texto o interfaces simples generadas con texto y caracteres especiales.
- **Interfaz gráfica de usuario** (Graphic User Interface, **GUI**): Permiten comunicarse con la computadora de forma rápida e intuitiva representando gráficamente los elementos de control y medida.
- **Interfaz natural de usuario** (Natural User Interface, **NUI**): Pueden ser táctiles, representando gráficamente un "panel de control" en una pantalla sensible al tacto que permite interactuar con el dedo de forma similar a si se accionara un control físico; pueden funcionar mediante reconocimiento del habla, como por ejemplo Siri; o mediante movimientos corporales (como es el caso del antiguo Kinect o las actuales Vision Pro de Apple).

En la Wikipedia está la historia de las interfaces gráficas de las que aquí se va a hacer un resumen.

2.1. Interfaz de línea de comandos

La interfaz de línea de comandos o interfaz de línea de órdenes (en inglés: *command-line interface*, **CLI**) es un tipo de interfaz de usuario de ordenador que permite a los usuarios dar instrucciones a algún programa informático o al sistema operativo por medio de una línea de texto simple.

Las CLI pueden emplearse interactivamente, escribiendo instrucciones en alguna especie de entrada de texto, o pueden utilizarse de una forma mucho más automatizada (archivo batch), leyendo órdenes desde

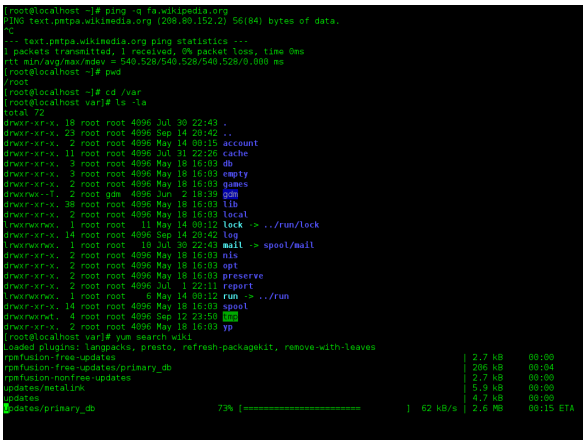
un archivo de scripts.

Esta interfaz existe casi desde los comienzos de la computación, superada en antigüedad solo por las tarjetas perforadas y mecanismos similares. Los ordenadores tenían conectadas "terminales tontas" que permitían enviar comandos al ordenador.

Terminal VT100. Fuente: Wikipedia

Hoy en día la línea de comandos se utiliza a través de programas denominados "emulador de terminales" que ejecutan un intérprete de comandos (llamado "shell"). Estos intérpretes, dependiendo del sistema operativo se pueden elegir, por ejemplo:

- En **Windows** se puede hacer uso del antiguo **cmd** o el más nuevo y mejorado **PowerShell**. También se puede utilizar otros, pero no vienen de serie instalados (como **bash**)
- En sistemas **GNU/Linux** el intérprete más utilizado y que está instalado en la gran mayoría de distribuciones es **bash**. En los últimos años ha ganado mucho mercado **zsh** gracias al *framework* Oh-my-zsh.
- En sistemas MacOS el intérprete por defecto es **zsh**.



Interfaz BASH. Fuente: Wikipedia

Los conceptos de CLI, shell y emulador de terminal **no son lo mismo** ya que CLI se refiere al paradigma, mientras que un shell o un emulador de terminal son programas informáticos específicos, que usualmente en conjunto implementan la CLI. **Sin embargo, los tres suelen utilizarse como sinónimos**.

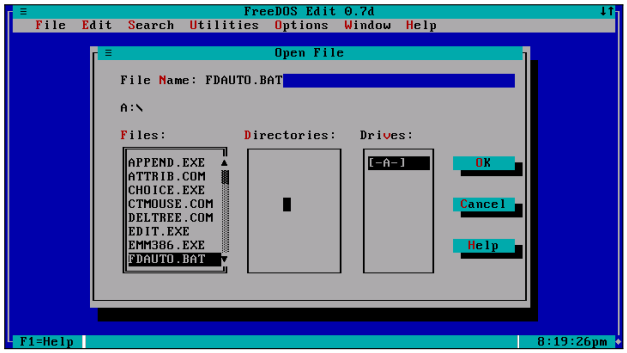
2.2. Interfaz gráfica de usuario

La interfaz gráfica de usuario, conocida también como **GUI** (del inglés *graphical user interface*), es un programa informático que actúa de interfaz de usuario, en la que hoy en día utilizando un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz.

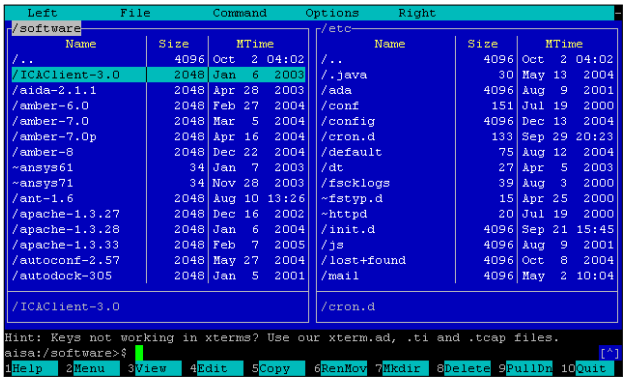
Su función principal es proporcionar un entorno visual sencillo para permitir la comunicación con el sistema operativo de un ordenador.

2.2.1. Interfaz en modo texto

Las interfaces gráficas no nacen con los escritorios, si no que ya en pasos anteriores existen librerías que simulan ventanas o entornos haciendo uso de caracteres de texto y colores.



FreeDos Edit. Fuente: Wikipedia



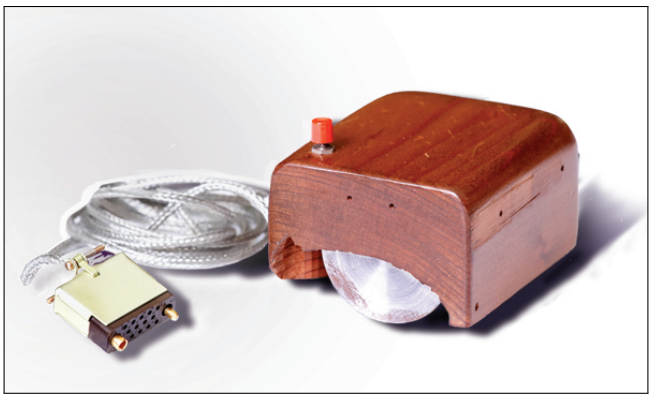
Midnight Commander. Fuente: Wikipedia

Hay que tener en cuenta que estos interfaces se solían utilizar en máquinas con muy pocos recursos y con sistemas operativos en los que no existía un interfaz gráfica.

2.2.2. Primeras investigaciones y desarrollos

El proyecto de “Aumento del Intelecto Humano” de Doug Engelbart en el Stanford Research Institute (en Menlo Park, EE. UU.) en la década de los 60 desarrolló el oN-Line System, que incorporaba un cursor manejado con un ratón y múltiples ventanas usadas para trabajar con hipertexto.

Mucha de la investigación inicial estuvo basada en como los niños pequeños aprenden. Por lo que el diseño se basó en las primitivas infantiles de coordinación mano-ojo, en lugar de usar lenguajes de comandos.

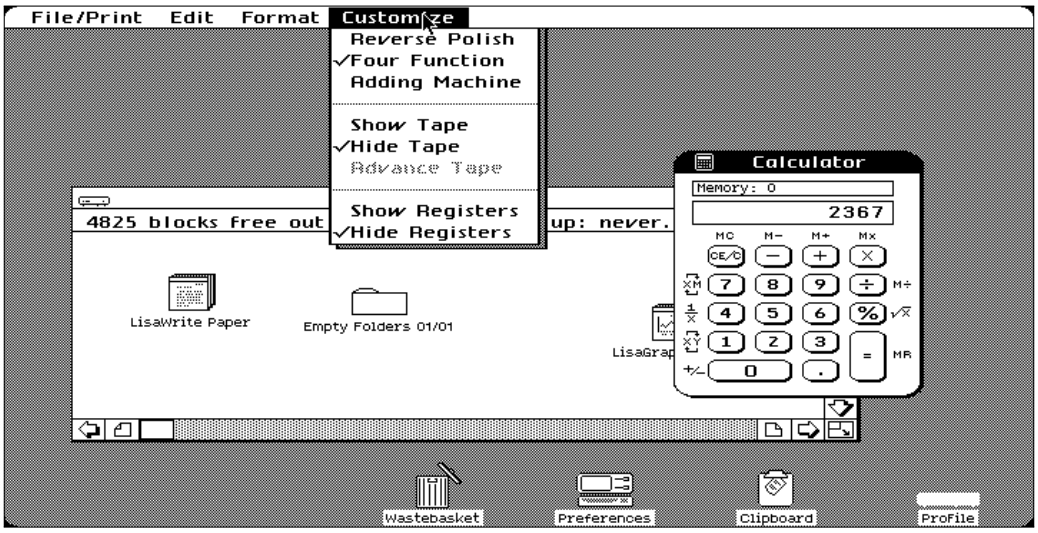


El primer prototipo de un ratón. Fuente: Wikipedia

El GUI fue inicialmente desarrollado en Xerox PARC (*Palo Alto Research Center*) por Alan Kay, Larry Tesler, Dan Ingalls y algunos investigadores más. Usaba ventanas, iconos y menús, incluyendo el primer menú desplegable fijo, para dar soporte a comandos como abrir ficheros, borrar y mover ficheros, etc.

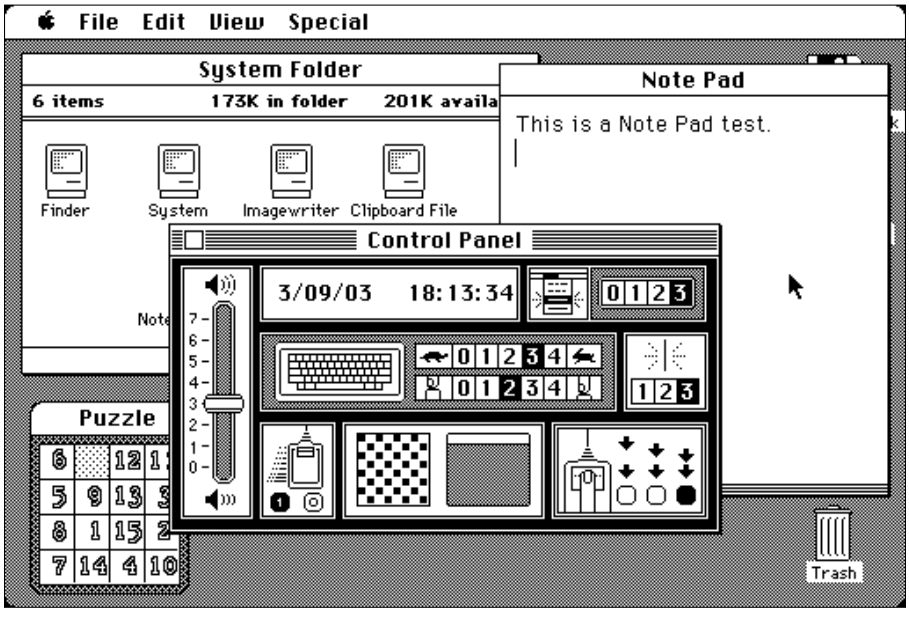
En 1981 Xerox presentó un producto innovador, el Star, incorporando muchas de las innovaciones de PARC. Aunque no fue un éxito comercial, el Star influenció de manera importante los futuros desarrollos, por ejemplo en Apple, Microsoft y Sun Microsystems.

En 1979, el equipo encabezado por Steve Wozniak, Steve Jobs y Jef Raskin (junto con antiguos trabajadores de Xerox) continuaron con las ideas vistas en Xerox PARC para lanzar en 1983 el ordenador "Apple Lisa", el primer ordenador con interfaz gráfico. No fue un éxito en ventas, pero inició la era de lo que se podrían denominar "los sistemas operativos con escritorio".



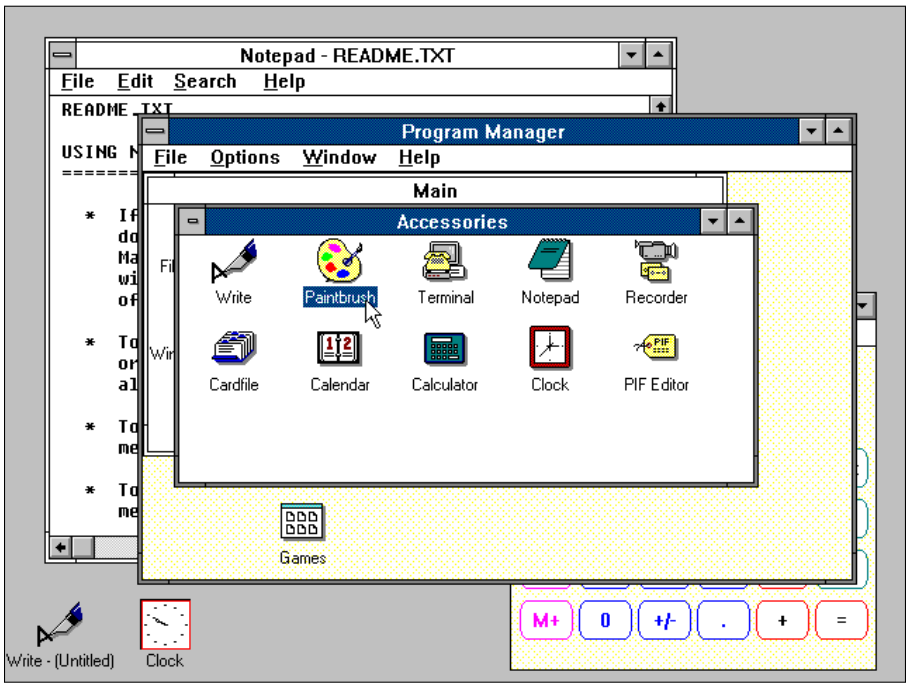
El interfaz gráfico de LisaOS 1.0. Fuente: GUI Gallery

En 1984 Apple lanzó el Macintosh, el cual presentó una evolución del sistema operativo de Lisa, llamado "System", en el que se puede apreciar partes que apenas han cambiado desde entonces.



El interfaz gráfico del Macintosh. Fuente: GUI Gallery

Al año siguiente Microsoft sacaría Windows 1.0, que era un GUI para su sistema operativo MS-DOS, pero no sería hasta la versión de Windows 3.0 de 1990 en la que su popularidad explotó.



El interfaz gráfico de Windows 3.0. Fuente: GUI Gallery

2.3. Uso masivo del escritorio

Con la llegada de la década de 1990, en la que los ordenadores personales (PC, del inglés *personal computer*) se popularizan y empiezan a llegar a las casas y negocios, hizo que se creara un mercado de usuarios sin conocimientos avanzados que requerían de facilidades de uso.

El "boom" comenzó con el Windows 3.11, pero sobre todo con Windows 95, que consiguió ocultar el núcleo del sistema operativo subyacente (MS-DOS) para mostrar directamente el interfaz gráfico desde el inicio.

El interfaz gráfico de Windows 95 en el primer arranque. Fuente: GUI Gallery

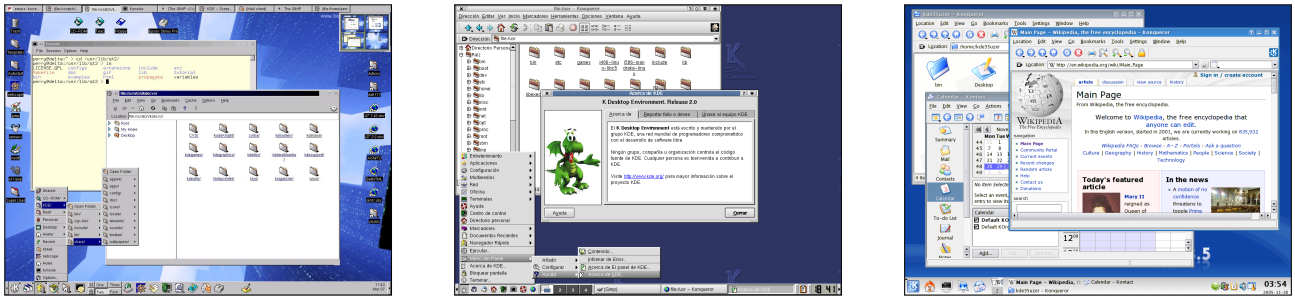
Apple compra a NeXT (empresa fundada por Steve Jobs) y utiliza su sistema operativo como los cimientos de Mac OS 8. Posteriormente cambiarían la arquitectura del sistema operativo para ser de estilo UNIX y así presentar Mac OS X en 2001.

Mac OS X 10.1 en 2001. Fuente: GUI Gallery

Durante estos años y principios de los 2000, los interfaces de usuario van evolucionando a medida que evoluciona la potencia de los ordenadores, pero siguiendo la misma tónica de simular un escritorio real en el ordenador.

Se anuncian entornos de escritorio libres como Gnome y KDE, que evolucionan en paralelo dentro de mundo de GNU/Linux llegando a crearse casi una guerra entre ellos y entre los usuarios (Enlace 1, Enlace 2).

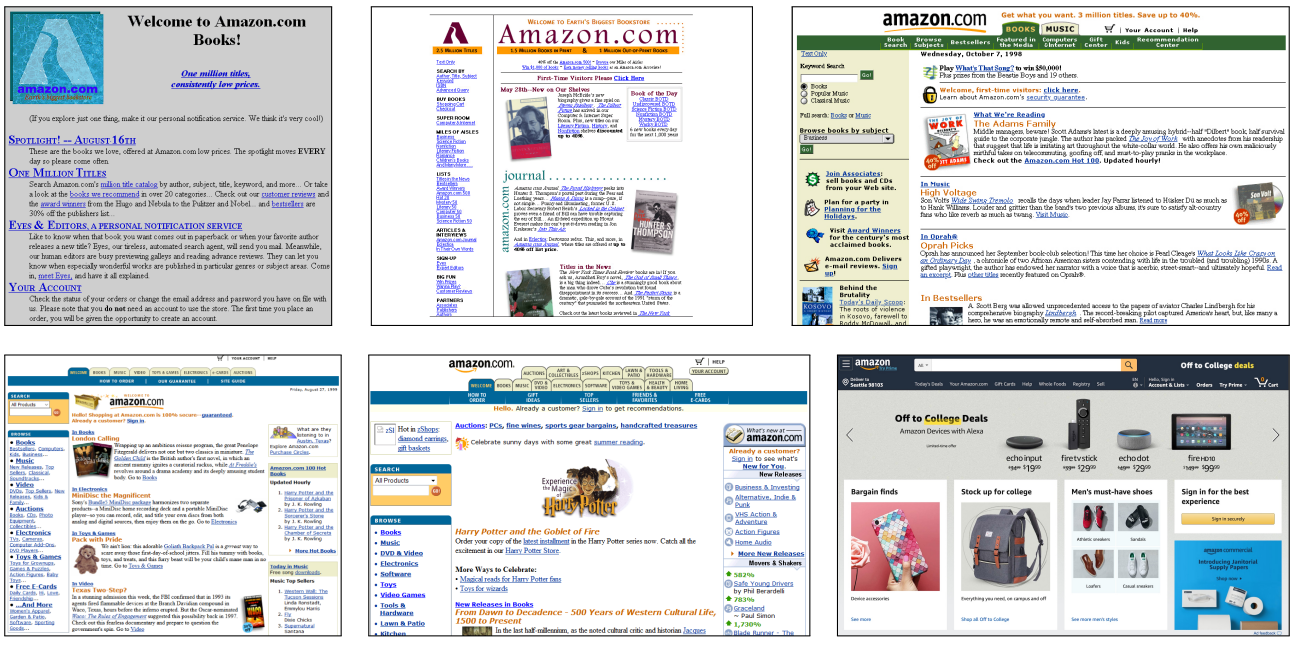
Cada sistema operativo y cada sistema de escritorio realizaba sus propias evoluciones que si tenían calado eran “copiadas” en los otros, generando también ciertas “modas” en colores, iconos o efectos de ventanas.



Evolución de KDE en 1998, 2000 y 2005. Fuente: Wikipedia

2.4. Interfaces web

Entre finales de los 90 y principios de los 2000, el auge de internet en todo el mundo hace que los portales web también sufran una gran evolución en sus interfaces.



Evolución de la web de Amazon. Fuente: VersionMuseum

También hay que tener en cuenta que las versiones HTML y CSS de la época no era tan avanzado como lo son hoy, y que las posibilidades de ejecución de javascript tampoco era tan potente como lo es hoy en día.

Las interfaces pueden verse limitadas por la propia tecnología

Información

Las interfaces pueden verse limitadas por las tecnologías del momento

Por otro lado, es habitual que la imagen de la compañía cambie, ya sea por cambios de logotipo, colores corporativos, moda en el tono de los colores... y todo eso también hace que los interfaces sufran modificaciones.

Evolución del logotipo de Twitter/X.

2.5. Interfaces móviles

En el caso de los teléfonos móviles, debido a que no existía un sistema estándar de visualización, cada fabricante iba creando su propio interfaz a medida que las pantallas iban creciendo en tamaño, resolución, colores...

Los primeros móviles carecían de pantalla, posteriormente comenzaron a tener pequeñas pantallas LCD monocromas, en algunos casos con una única línea de texto.

Ericsson T10 y Nokia 3310

Posteriormente comenzaron a tener pantallas a color y cada compañía crea su propio sistema operativo. Existe un intento de crear un sistema operativo entre varias compañías llamado Symbian, pero al final es Nokia la desarrolladora y la que más uso le da.

Durante los primeros años de la década de los 2000 también se ponen de moda las PDA (*personal digital assistant*, asistente digital personal u ordenadores de bolsillo), que normalmente hacían uso de un lapiz para hacer funcionar la pantalla táctil (resistiva) que tenían.

Palm

Todo cambió con la llegada del iPhone en el 2007 y los primeros *smartphones*, donde las pantallas táctiles capacitativas (se manejan con el dedo) cobran protagonismo. Estas pantallas aparte de ser táctiles también permiten gestos con varios dedos, dando lugar a nuevos efectos de interactuar con el interfaz.

iPhone 1ª generación y HTC Desire.

Apple se decanta por un único botón mientras que los sistemas operativos Android (como el HTC Desire del 2009) tienen al menos cuatro para manejar ciertas tareas del sistema y de la aplicación que se está usando en el momento.

Con estos cambios, los sistemas operativos y las interfaces sufren un cambio radical, donde buscan la facilidad del usuario y aparecen los "gestos" como nueva manera de interactuar con el interfaz:

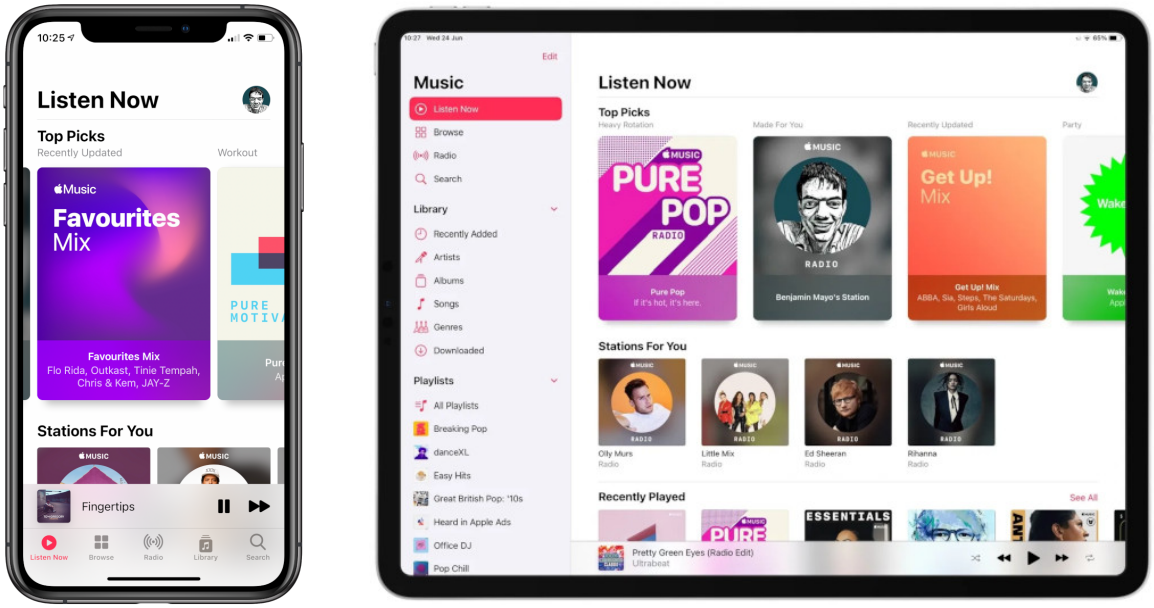
- **Tap**: tocar la pantalla (similar al "click" con el ratón)
- **Double tap**: doble toque, que dependiendo de la aplicación puede hacer un zoom específico.
- **Scroll**: presionando con el dedo y subiendo para arriba o para abajo se hace scroll sobre el interfaz.
- **Flick**: si se hace un gesto rápido hacía arriba o hacia abajo se hace scroll, mientras que si es a izquierda o derecha el movimiento dependerá de si hay más pantalla.
- **Pinch in**: hacer zoom acercando lo que se quiere ver.
- **Pinch out**: hacer zoom alejando el objeto.

Con todo ello, surge un nuevo "problema" que hasta este momento no se había planteado en los interfaces, que es **la rotación del dispositivo**. Esta rotación debe hacer que la aplicación que se está ejecutando se adapte a la nueva posición del terminal, pasando de vertical a horizontal o viceversa.

Vista vertical y horizontal de la app Music de iOS

Como evolución de estos dispositivos surgen las *tablets* en donde se sigue la misma filosofía de gestos, pero las pantallas táctiles de mayor tamaño permiten tener más espacio para información. En lugar de usar los interfaces de escritorio, evolucionan los interfaces de los móviles.

Al tener más espacio, no se utilizan las aplicaciones de móviles y se “escalan”, si no que surgen nuevas posibilidades para que los interfaces dispongan de nuevos menús o lugares donde añadir más información.



Vista de la app Music de iOS en iPhone e iPad. Fuente: 9to5mac

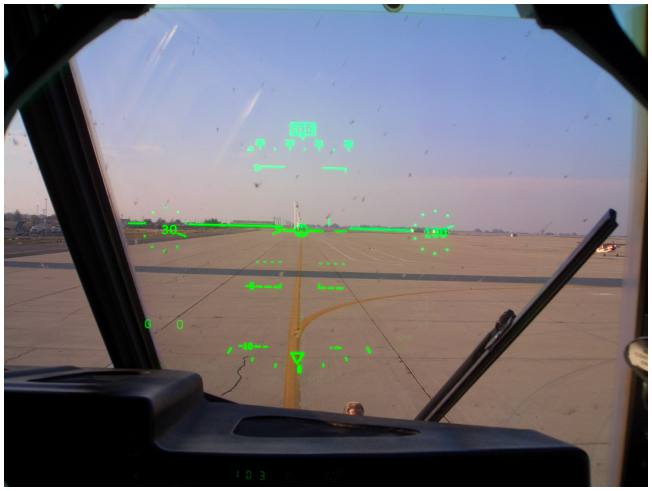
¡Atención!

Al diseñar un interfaz tenemos que tener en cuenta el dispositivo, las dimensiones y el uso que se le va a dar para mejorar la experiencia de usuario.

2.6. Realidad virtual y realidad aumentada

El siguiente paso en el desarrollo de interfaces se encuentra en la conocida como realidad aumentada y en la realidad virtual. Aunque puedan parecer términos que están ahora de moda, su estudio y evolución lleva décadas fraguándose.

Los primeros pasos que se dieron en este sentido son los denominados "head-up display", que son pantallas transparentes que presentan información al usuario dejando ver lo que está en la parte trasera de la pantalla. De esta manera, el usuario no tiene que girar la cabeza para obtener la información.



Ejemplo de interfaz *head-up* en un avión. Fuente:Wikipedia

En algunos casos esta pantalla puede estar entre el usuario y el salpicadero, integrada en el salpicadero o también como una pantalla puesta en el casco del piloto y cerca del ojo.

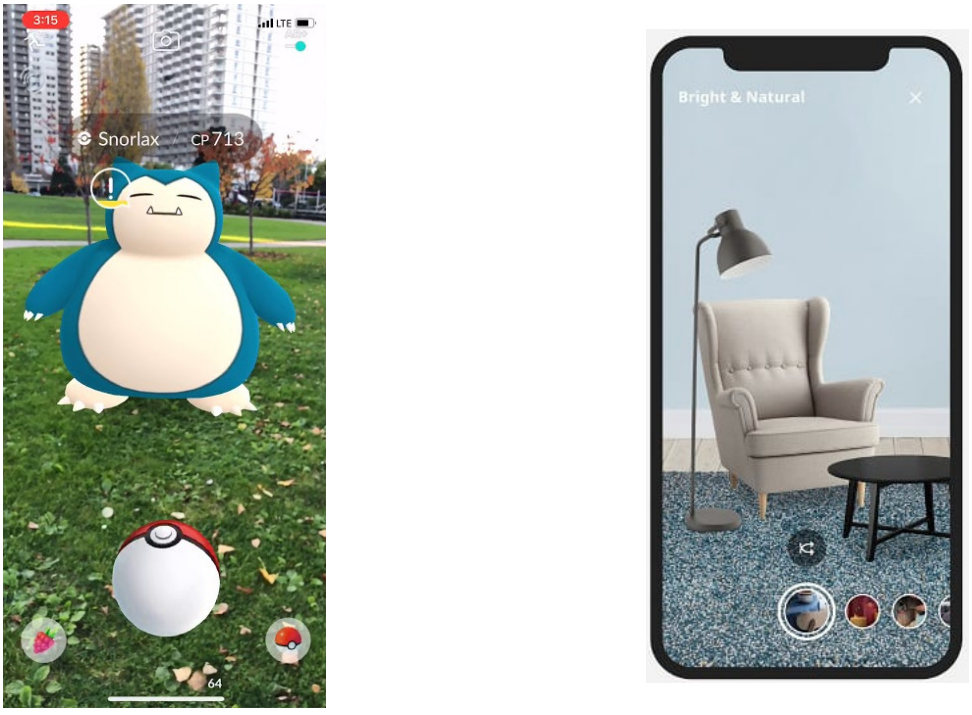
Por otro lado, tenemos **la realidad virtual, que es un entorno simulado que da una sensación de inmersión al usuario ofreciéndole la posibilidad de moverse en él**, y en algunos casos interactuar con elementos del entorno.

Para ello, suele ser habitual contar con unas gafas que nos mostrarán el entorno virtual, y también dispondremos de unos mandos para poder interactuar con el entorno.

Casco y mandos del HTC Vive

Para poder movernos por el entorno, algunos de estos sistemas hacen uso de *trackers*, de esta manera nos podremos desplazar por el entorno virtual sin tener que usar los mandos.

El siguiente paso es la conocida como **realidad aumentada**, en el que se trata de integrar elementos virtuales en la vida real. No es algo nuevo, ya que lleva años existiendo en juegos y aplicaciones.



Ejemplos de realidad aumentada con el móvil

Cn la idea de que la realidad aumentada sea de uso cotidiano han empezado a evolucionar los sistemas para que poco a poco esto vaya fraguando, aunque el coste todavía sigue siendo elevado. A día de hoy, tampoco está claro cuál será el beneficio y el rendimiento que tendrá.

En el caso de las Apple Vision Pro prometen una experiencia de realidad aumentada utilizando únicamente gestos de las manos para moverse a través de su interfaz.

Apple Vision Pro

3. Curiosidades, prototipos y ciencia ficción

En este apartado se van a añadir algunos ejemplos de interfaces que son reales, pero que no dejan de ser meras curiosidades, ejemplos de prototipos de interfaces que de momento no han llegado a nada, y la visión en algunos casos que nos da las películas de ciencia ficción.

- Demonstración de 1982 de posibles usos de tecnología táctil en pantallas. Realmente la pantalla no es táctil, tiene sensores alrededor de ella. Vídeo.
- En Jurassic Park aparece un interfaz para navegar un sistema de ficheros en 3D. It's UNIX!.
- La película Minority Report del 2002 muestra un interfaz que se utiliza con las manos. Vídeo.
- En 2003 Microsoft muestra un vídeo de lo que será su próximo Windows, llamado Longhorn. Este interfaz nunca llegó a la versión final, aunque sí que hubo versiones de demonstración que se parecían un poco, pero la versión final que se llamó Vista nunca llegó a ser como los primeros conceptos.
- En 2006, con la aceleración de OpenGL por *hardware* en las tarjetas gráficas, en las distribuciones GNU/Linux nacen los gestores de ventanas con diseños 3D real. Hubo varias alternativas, ya que surgieron varios "forks" de distintos proyectos, pero los más conocidos fueron Compiz y Beryl. Vídeo.
- En 2006 en una charla TED Jefferson Han realiza una demonstración de pantalla multitáctil con muchos gestos que posteriormente se volverían el día a día con los *smartphones*. Charla TED y Vídeo de demonstración.
- En 2009 Microsoft presenta un vídeo con el concepto de una tablet plegable llamada Courier. Vídeo 1, Vídeo 2. El iPad todavía no se había presentado.
    - Microsoft tiene varias páginas con conceptos de interfaces para tablets plegables en su web. Enlace 1, Enlace 2.
- Las películas de Iron Man han ofrecido muchas escenas donde aparecen interfaces muy espectaculares, a nivel holográfico y de realidad aumentada. Vídeo.

II

Android UI

1. Interfaces con Android UI

A la hora de crear interfaces para dispositivos Android a través de Android Studio, dependiendo de la aplicación que estemos creando, pueden ser generadas de dos maneras distintas:

- **Diseño de vista en XML**: Son generadas a través de ficheros XML utilizando un editor en el que podremos colocar distintos elementos, restricciones… Para el correcto funcionamiento y compilación del proyecto, estos ficheros deben de cumplir con el estándar XML y ser ficheros válidos. Era el sistema original a la hora de crear vistas.
- **Jetpack Compose**: es un *framework* Kotlin para crear interfaces de usuario de manera declarativa. En este caso se programa los componentes que va a tener el interfaz.

En un mismo proyecto puede coexistir vistas creadas de distintas maneras, por lo que en un proyecto antiguo que se usó XML, podemos ir migrando vistas al formato Compose si usamos Kotlin.

1.1. Diferencias entre ambos modelos

A la hora de elegir qué sistema elegir, dependerá de los conocimientos que tengamos en ambos, pero a continuación se muestra una pequeña tabla comparativa:

- **XML**:
  - Ventajas:
    - **Separación de conceptos**: La vista se diferencia cláramente de la lógica
    - Editor **WYSIWYG**: Android Studio incorpora un editor visual que nos permite colocar los componentes de la vista. Puede ser más intuitivo para algunos desarrolladores
    - **Reciclaje**: Se pueden generar plantillas/*layouts* XML que pueden ser utilizados en distintos *activities*.
  - Desventajas:
    - **Limitado**: A la hora de generar interfaces complejos, el formato XML tiene limitaciones que tienen que ser subsanado con código Java/Kotlin.
    - ***Verbosity***: Los ficheros XML pueden ser complejos y tener redundancias. Esto puede hacer que se complique a la hora de mantener vistas complejas.
- **Jetpack Compose**
  - Ventajas:
    - **Sintaxis declarativa** que facilita la creación de UI complejas
    - **Tipado seguro**: Como es código Kotlin, se reduce la posibilidad de cometer errores de tipado y que haya errores durante la ejecución.
    - **Modularidad**: Jetpack Compose promueve la creación de componentes de interfaz que se puedan reutilizar. Esto facilita el mantenimiento de la aplicación y la escalabilidad.

- **Vista previa "en vivo"**: Se puede ver una vista previa a medida que se va generando la vista en código.

- Desventajas:
  - La **curva de aprendizaje** puede ser pronunciada al inicio.
  - **Compatibilidad**: A día de hoy no debería ser un problema, pero sólo se puede usar con **Android 5.0 (API level 21)** o superior.
  - Sólo se puede usar con Kotlin.

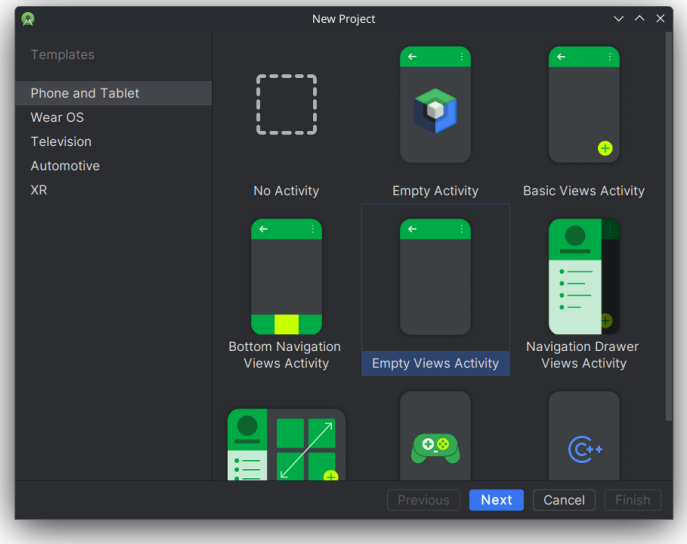
2. Diseño de vistas en XML con Android Studio

A continuación se explica la creación de un proyecto donde por defecto se nos va a crear el ***MainActivity*** utilizando una vista en formato XML, conocidas como *Android View*, que es el sistema antiguo de creación de vistas.

Los ficheros de las vistas se encuentran por defecto en la ruta `src/main/res/layout` de nuestro proyecto. No es habitual editar estos ficheros fuera de Android Studio, pero hay que tener en cuenta dónde se almacenan.

2.1. Creando nuestro primer proyecto

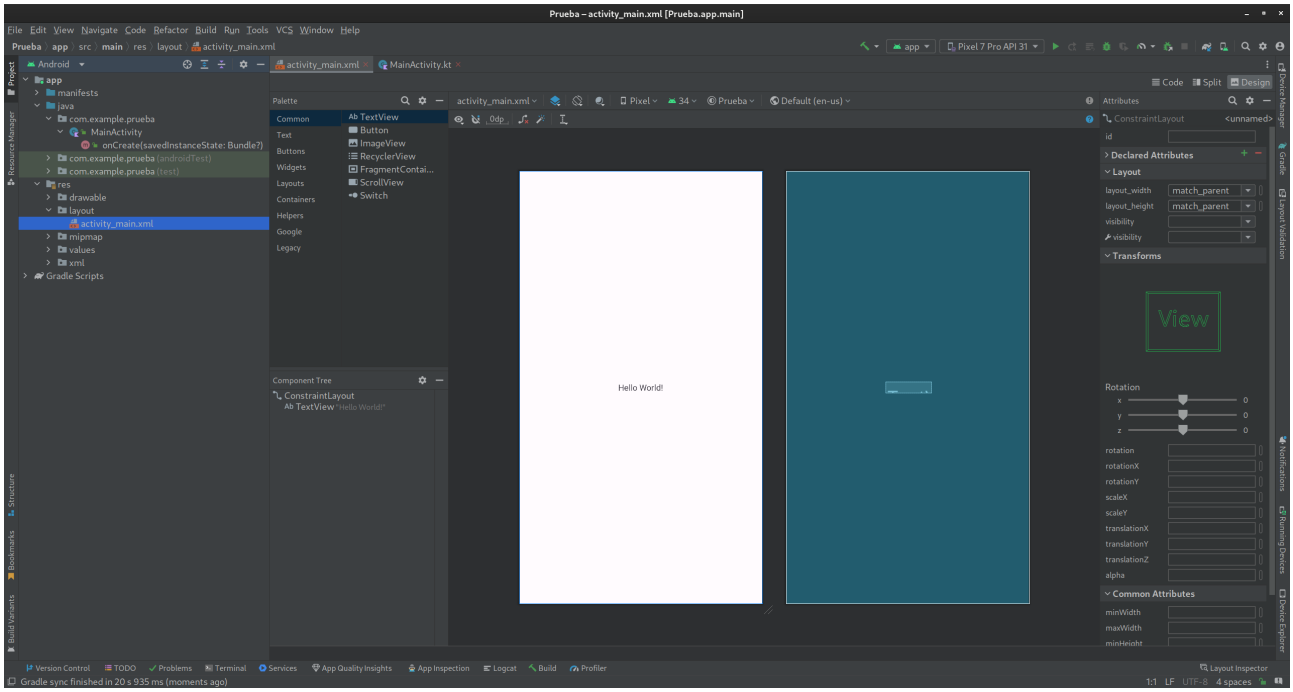
Una vez abierto Android Studio vamos a crear un nuevo proyecto y vamos a elegir "Empty Views Activity", de esta manera el proyecto que nos va a generar va a tener una primera vista con un texto centrado



En el siguiente paso elegimos el nombre del proyecto, la ruta, el lenguaje de programación y el SDK que vamos a utilizar. Con las opciones elegidas, se empezará a generar el proyecto y los ficheros necesarios para poder hacer funcionar la aplicación.

Si desplegamos en el menú de la izquierda (con la opción "Android" seleccionada) hasta llegar a la ruta

`app/res/layout`, veremos que tenemos un fichero que se llama `activity\_main.xml`, que al hacer click obtendremos una vista similar a esta:



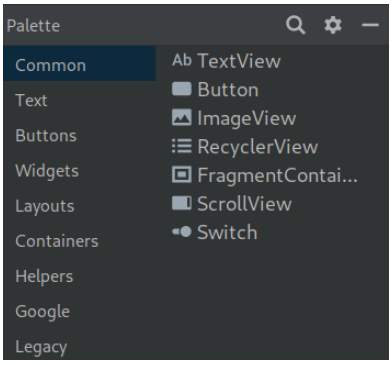
2.2. Vista “diseño”, “blueprint” y código

A la hora de interactuar con las vistas tendremos tres opciones:

- **Vista diseño**: Es lo más cercano a lo que el usuario final va a ver. De esta manera podremos ver los distintos elementos que tiene la interfaz y con el contenido que se le haya dado.
- **Vista blueprint**: También conocida como “cianotipo” o vista de plano. Simula los planos de ingeniería y en esta vista sólo veremos de qué tipo es cada elemento. Esta vista es adecuada para abstraernos del contenido y centrarnos en el diseño.
- **Código XML**: Es el contenido del fichero XML de la propia vista. Podremos realizar modificaciones cambiando el texto. Esta vista es la idónea si queremos cambiar algún parámetro concreto que ya está prefijado y queremos ir a tiro hecho.

2.3. Componentes en el interfaz

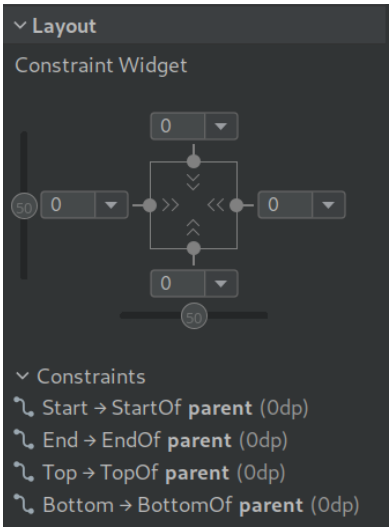
A la hora de crear nuestro interfaz Android Studio tiene una serie de componentes que podremos utilizarlos diréctamente arrastrando los componentes al interfaz.



Debe quedar claro que cada uno de ellos contará con sus características propias, por lo que es importante hacer uso de la documentación oficial a la hora de utilizar cada uno de ellos.

2.4. Layouts y constraints

Con la vista inicial, si seleccionamos el texto “Hello World!” veremos que tiene una configuración en el apartado ***Layout*** como el siguiente:



Se pueden identificar 4 *constraints* que hacen referencia a los distintos puntos del componente:

- ***Start***: Lado izquierdo del componente.
- ***End***: Lado derecho del componente.
- ***Top***: Parte superior del componente.
- ***Bottom***: Parte inferior del componente.

De esta manera, cada parte del componente tendrá un *constraint* que estará asociado, en este caso, a una parte del *layout* padre que lo contiene. Si miramos el código fuente del fichero XML, se puede apreciar cómo cada uno de estos aparece escrito:

Código XML de un TextView

```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

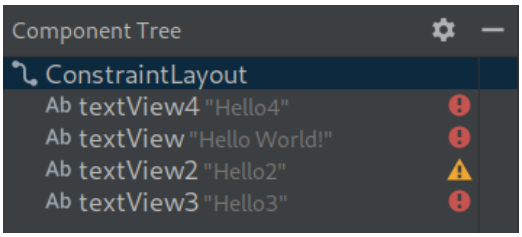
Si añadimos al interfaz un botón, en la parte baja de la pantalla, centrado y en la posición que nos interese en la vista vertical, veremos que a la hora de ejecutar la vista en el simulador deja de aparecer en la posición que lo hemos indicado.

¡Cuidado!

Los atributos “layout_editor” del XML sólo sirven para la vista del editor, no para la vista final. Para colocar los elementos en su posición correcta hay que usar los constraints.

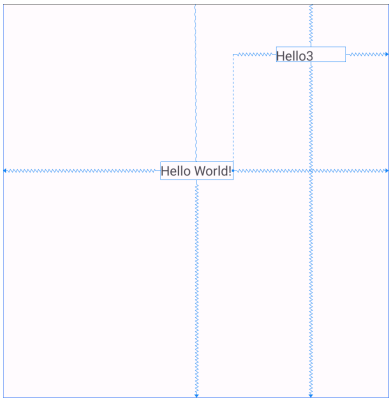
Es por eso, que para cada componente que queramos añadir a la vista deberá tener sus *constraints* haciendo

referencia a otros componentes o al *layout* que lo contiene. En caso de que no hayamos añadido uno, o algo en la vista no sea correcto, nos aparecerá en el interfaz:



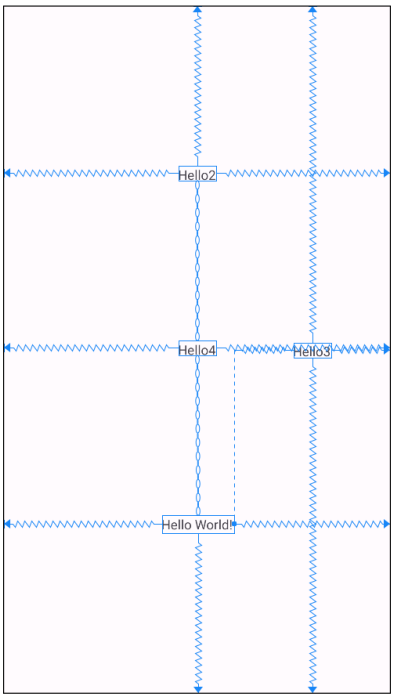
Los componentes pueden estar asociados a otros componentes, pero no tienen por qué estar asociados entre ellos. Por ejemplo, la siguiente vista tiene a nivel horizontal podemos apreciar lo siguiente:

- El texto “Hello World” está asociado a ambos lados del layout que lo contiene, y por eso aparece centrado.
- El componente con texto “Hello3” es más ancho que el texto que lo contiene (atributo ***layout_width***). En el lado izquierdo está asociado al otro componente mientras que en el lado derecho a la vista, haciendo que quede horizontalmente entre ambos.



2.4.1. Cadenas

Las cadenas sirven para enlazar un grupo de componentes (o vistas) de manera bidireccional entre ellos. Se pueden realizar cadenas de manera vertical y horizontal. Ejemplo de cadena vertical con los 3 elementos que están centrados.

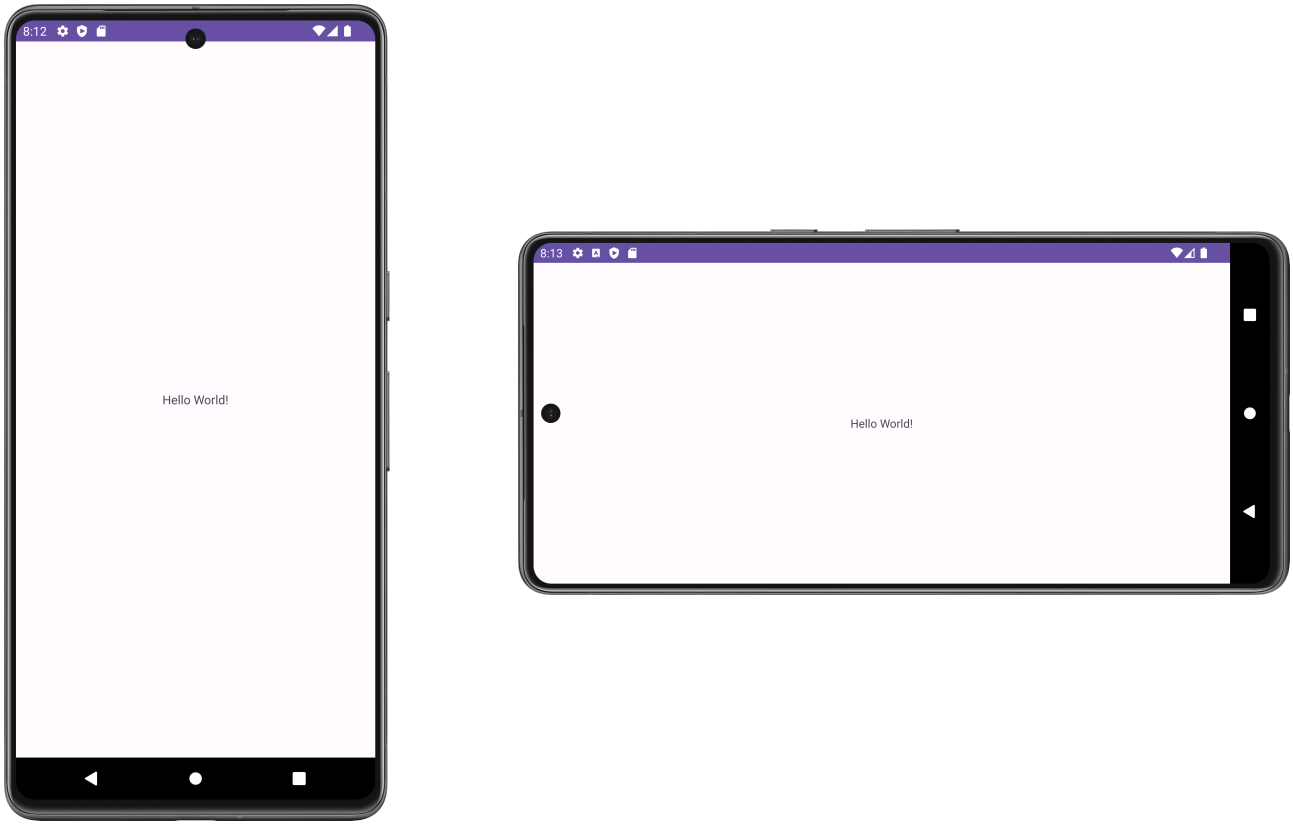


2.5. Vista vertical y horizontal

Nuestros dispositivos pueden rotar de posición, y por tanto los componentes que aparecen en pantalla deberían poder ajustarse a la nueva posición de la misma.

Por eso, a la hora de diseñar el interfaz, deberemos pensar qué hacer con la posición de los componentes: si se desplazan de posición, si se ocultan de alguna manera, si se convierten en un menú desplegable...

Por defecto, al rotar el dispositivo, los componentes se moverán teniendo en cuenta el ***layout*** y los ***constraints*** que tienen configurados. En la vista creada al crear el proyecto, al rotar el dispositivo sucederá lo siguiente:

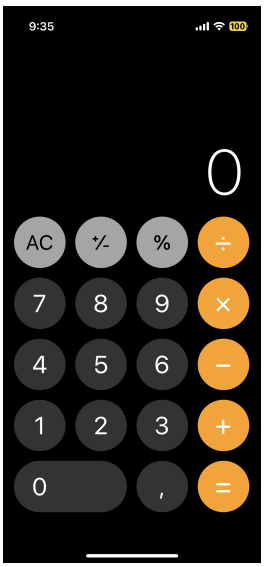


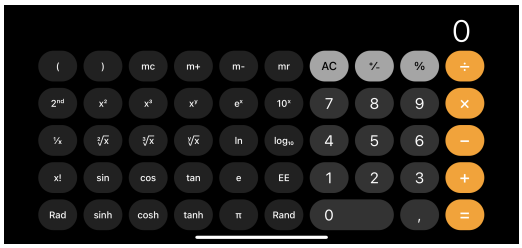
Tal como se puede ver, el texto “Hello World!” permanece centrado respecto a la vista debido a que el texto tiene asociados unos constraints contra el resto del interfaz, haciendo que siempre esté centrado.

2.5.1. Diferenciando las vistas

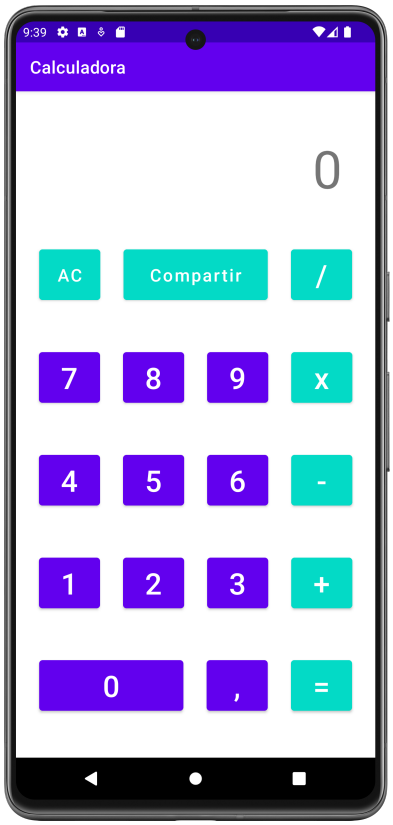
A la hora de tener distintas vistas para la posición del dispositivo, es posible que no nos interese tener todos los elementos, o que estos no estén situados en la misma posición.

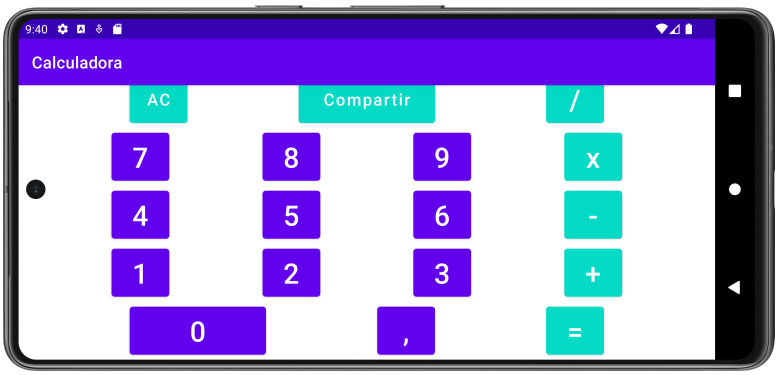
Por ejemplo, la calculadora de iOS a la hora de rotar el dispositivo, la calculadora ofrece más opciones convirtiéndola en una calculadora científica.





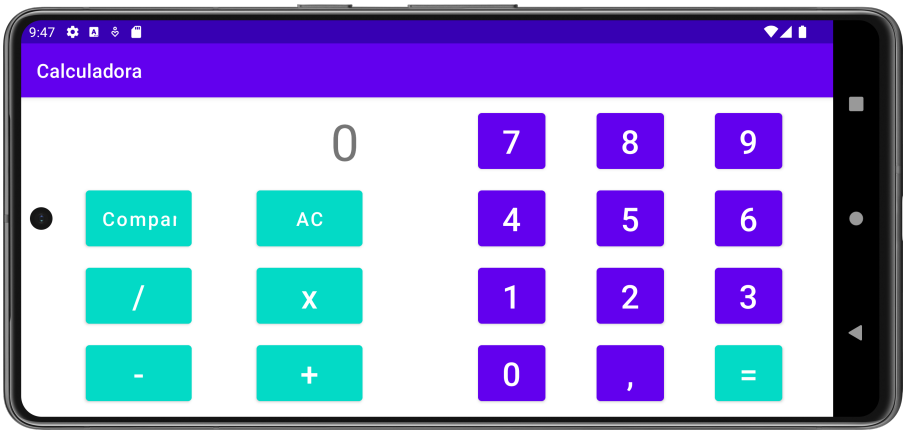
Si creamos una vista sin realizar modificaciones para el modo "*landscape*", obtendremos un resultado similar a este:





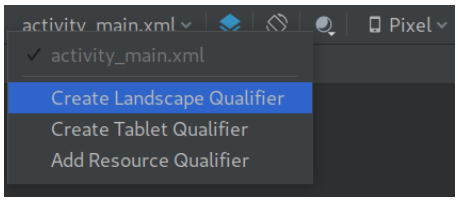
Como se puede ver, la colocación de los botones se mantiene igual en ambas vistas, pero hace que el modo apaisado no sea usable. Y aunque pudiésemos ver el resultado, la vista no tendría mucho sentido.

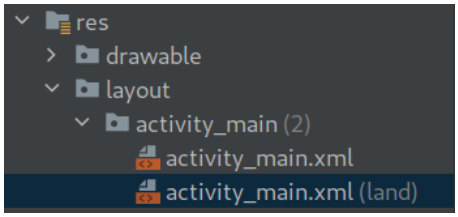
En cambio, si realizamos una composición distinta para la vista apaisada, obtendremos un resultado más óptimo, donde podremos tener los elementos colocados de una manera distinta, pero más usable.



Para tener vistas diferenciadas tendremos que crear una vista en modo apaisado sobre la vista que ya tengamos creada. Para ello vamos al editor de la vista, hacemos click en el desplegable y seleccionamos "*Create Landscape Qualifier*".

Esto nos creará en el árbol de nuestro proyecto una carpeta con el nombre de la vista original y dos ficheros: el original y otro donde podemos ver que le ha añadido "*land*" para identificar que es la versión apaisada.





Gracias a esto nos va a crear una vista nueva, con todos los elementos de la vista original, manteniendo los *constraints*. El nuevo fichero **xml** lo podremos encontrar en una nueva ruta que ha creado un directorio para las vistas apaisadas: `app/src/main/res/layout-land`.

2.6. Dando funcionalidad al interfaz

Aunque no se va a ahondar en cómo programar para Android, se va a añadir un pequeño ejemplo para entender cómo funciona un botón y un texto y cómo interactuar entre sí.

La idea es:

- Crear un interfaz con un botón y un TextView.
- Al pulsar el botón, el texto original del TextView desaparece y se incrementa en 1 para saber el número de pulsaciones que se han realizado.

Teniendo esto en cuenta, el código del fichero **MainActivity.kt** debería quedar de la siguiente manera:

Código Kotlin para hacer funcionar un botón

```kotlin
//... imports previos
import android.widget.Button
import android.widget.TextView
```

```
class MainActivity : AppCompatActivity() {

    var contador:Int = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        //... código ya existente

        val button:Button = findViewById(R.id.button)
        val text:TextView = findViewById(R.id.textView)

        button.setOnClickListener {
            contador = contador+1
            text.text= (contador).toString()
        }
    }
}
```

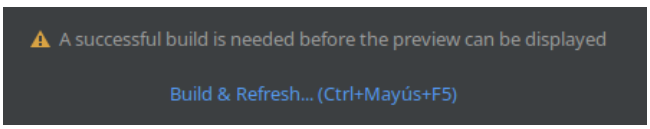
Pregunta

¿Qué pasa con los datos si rotamos la pantalla?

3. Diseño de vistas con Jetpack Compose

En este caso, el proyecto lo crearemos seleccionando la opción “Empty Activity”, lo que nos creará un proyecto donde por defecto se nos va a crear la clase ***MainActivity*** utilizando el nuevo sistema **Jetpack Compose**, para la creación de vistas, y por tanto haciendo uso de la sintaxis declarativa.

Al terminar de crearse el proyecto, si utilizamos en la vista el sistema *Split*, junto al código fuente veremos el mensaje que nos indica que debemos “construir” la vista y refrescar para ver el resultado en tiempo real. Al hacer click, se construirá la vista y a partir de ese momento tendremos una previsualización en tiempo real, tal como aparece a continuación:



3.1. Entendiendo la vista por defecto

La vista que nos ha generado el proyecto cuenta con una serie de apartados que hay que entender para poder realizar modificaciones y también saber qué está sucediendo en la vista previa.

3.1.1. Función que genera un *widget* de vista

Tal como se ha dicho previamente, *Compose* trata de generar vistas, **o partes de ella**, que puedan ser reutilizadas. En este caso, el asistente de creación del proyecto nos ha creado una función `fun Greeting(...)` que genera un pequeño componente de tipo **Text** que genera y visualiza un texto utilizando el párametro que se le pasa a la función, y un posible modificador (luego hablaremos de ellos).

Widget que genera un texto

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}
```

De esta función lo que nos interesa es entender la anotación `@Composable`, que es una anotación que todas las funciones de "componibilidad" deben tener. Este `@Composable` informa al compilador de Compose que esta función está diseñada para convertir datos en IU.

3.1.2. Vista previa de Compose

Una de las ventajas que tenemos a la hora de usar Compose es que podemos tener una vista previa a medida que vamos escribiendo el interfaz, sin necesidad de utilizar el emulador del terminal, lo que hace que **consumamos menos recursos**. La función que nos genera la vista previa tiene la anotación `@Preview`, a la que podremos añadir distintos parámetros.

Función que genera la vista previa

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    JetpackTheme {
        Greeting("Android")
    }
}
```

Tal como se ha visto previamente, tenemos una vista previa que nos muestra el "Hello Android", pero esta vista **no simula un interfaz real**. Esto no tiene por qué ser malo, ya que nos podemos centrar sólo en el interfaz/widget que estemos creando.

Podemos tener tantas vistas previas como nos interese, para diferentes dispositivos, modo claro/oscuro… Para generar nuevas vistas previas, debemos añadir una nueva anotación `@Preview` con los parámetros

que nos interese. Por ejemplo:

Nuevas vistas previas con distintos parámetros

```
@Preview(showBackground = true, showSystemUi = true,
    uiMode = Configuration.UI_MODE_NIGHT_YES)
@Preview(showBackground = true, showSystemUi = true,
    device = "id:pixel_fold")
@Preview(showBackground = true, showSystemUi = true,
    device = "spec:width=411dp,height=891dp,orientation=landscape")
```

En estos ejemplos se han añadido distintos parámetros a la anotación `@Preview`, que son:

- **showSystemUi**: nos muestra el interfaz como si se estuviese ejecutando en un dispositivo.
- **uiMode**: acepta cualquier valor de Configuration.UI_*. En el ejemplo hará uso del modo noche.
- **device**: podemos indicar un dispositivo concreto mediante su **id**
  - Podemos crear nuestro propio tipo de dispositivo con el parámetro **spec:**, en el que le podemos indicar la orientación también.

Existen más posibilidades de cómo generar vistas en la documentación oficial.

3.1.3. La vista en MainActivity

Tras todo lo anterior, es momento de entender qué sucede en la función principal de nuestra clase **MainActivity**, que es la función que se va a ejecutar al arrancar nuestra aplicación, al igual que en el anterior proyecto, **onCreate**.

MainActivity con código Jetpack Compose

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            JetpackTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "Android",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}
```

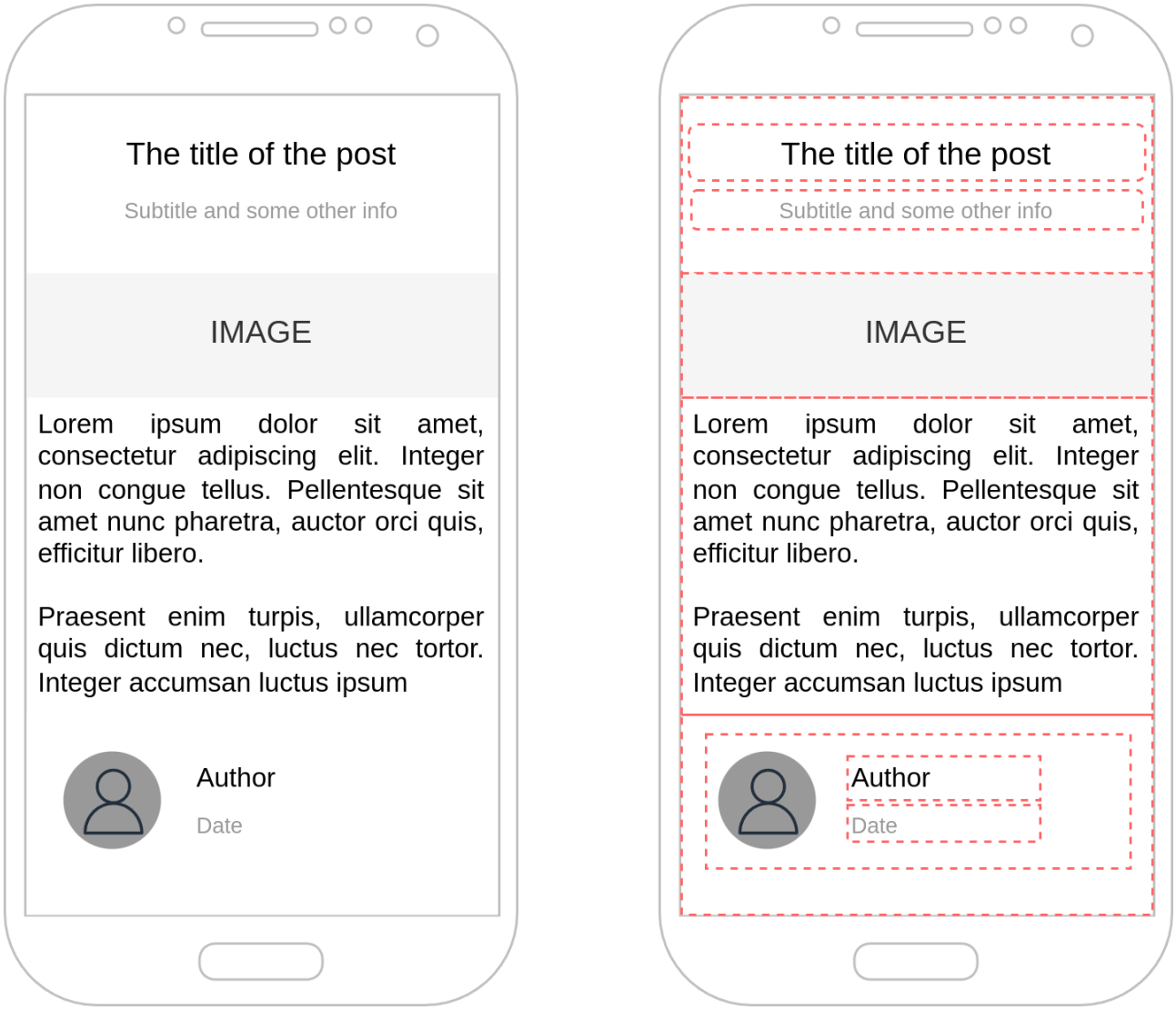
En esta función el bloque **setContent** define el diseño de la actividad, donde se llaman a las funciones que admiten composición. Estas funciones solo se pueden llamar desde otras funciones del mismo tipo.

Luego podemos ver que todo está también dentro de **JetpackTheme** que es una función que está declarada en el fichero `ui/theme/Theme.kt`. Esta función generada lo que hace es diferenciar si el dispositivo está en modo claro o modo oscuro, para hacer uso del tema que hayamos creado.

3.2. Disposición de elementos

A la hora de crear nuestro interfaz debemos pensar en la composición/disposición (*layout* en inglés) de los distintos elementos. En la documentación oficial hay un ejemplo similar al que se va a explicar aquí, por lo que es recomendable echar un vistazo.

Vamos a suponer el siguiente interfaz básico, donde a la izquierda aparece cómo quedaría el interfaz final, mientras que la disposición de los distintos elementos aparecen resaltados en la imagen de la derecha.



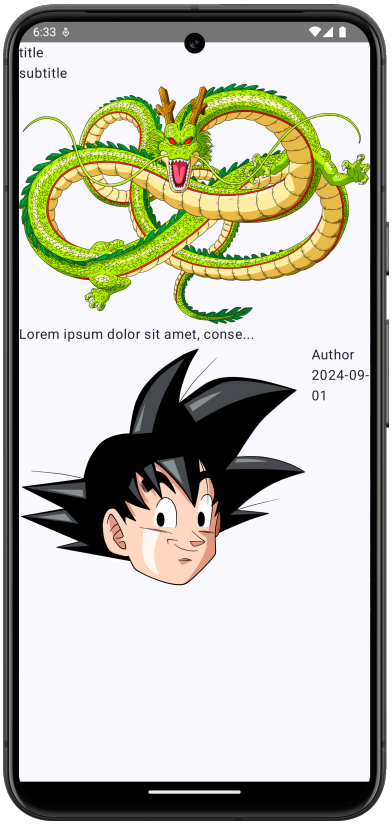
Vista real y vista diferenciando la disposición

Mirando la disposición diseñada, a la hora de convertirlo a programación declarativa debemos entender los distintos apartados que hemos creado y cómo se van a colocar los distintos componentes.

Una primera iteración en programación con Composer quedaría de la siguiente manera, donde se han colocado los componentes con contenido *hardcodeado*, por lo que la función creada siempre mostrará el mismo contenido:

Función con código Compose

```
@Composable
fun Post() {
    Column {
        Column {
            Text("title")
            Text("subtitle")
        }
        Image(
            painter = painterResource(R.drawable.dragon),
            contentDescription = null
        )
        Text("Lorem ipsum dolor sit amet, conse...")
        Row {
            Column {
                Image(
                    painter = painterResource(R.drawable.goku),
                    contentDescription = null
                )
            }
            Column {
                Text("Author")
                Text("2024-09-01")
            }
        }
    }
}
```



Tal como se puede ver en el código, se han utilizado las siguientes funciones para colocar los elementos:

- **Column**: pone todos los componentes hijos en una secuencia vertical.
- **Row**: para colocar los componentes hijos en una secuencia horizontal.

3.3. Modificadores de los elementos

La función creada previamente coloca los distintos elementos haciendo uso de las funciones de Composer, pero para tener una interfaz amigable y con el resultado final, debemos hacer uso de los modificadores para cada elemento. Este será el punto en el que decidiremos "cómo poner bonito" el interfaz.

¡Cuidado!

Los modificadores se pueden añadir a nivel de bloque o a nivel de elemento. Hay que tener cuidado con las prioridades de los modificadores.

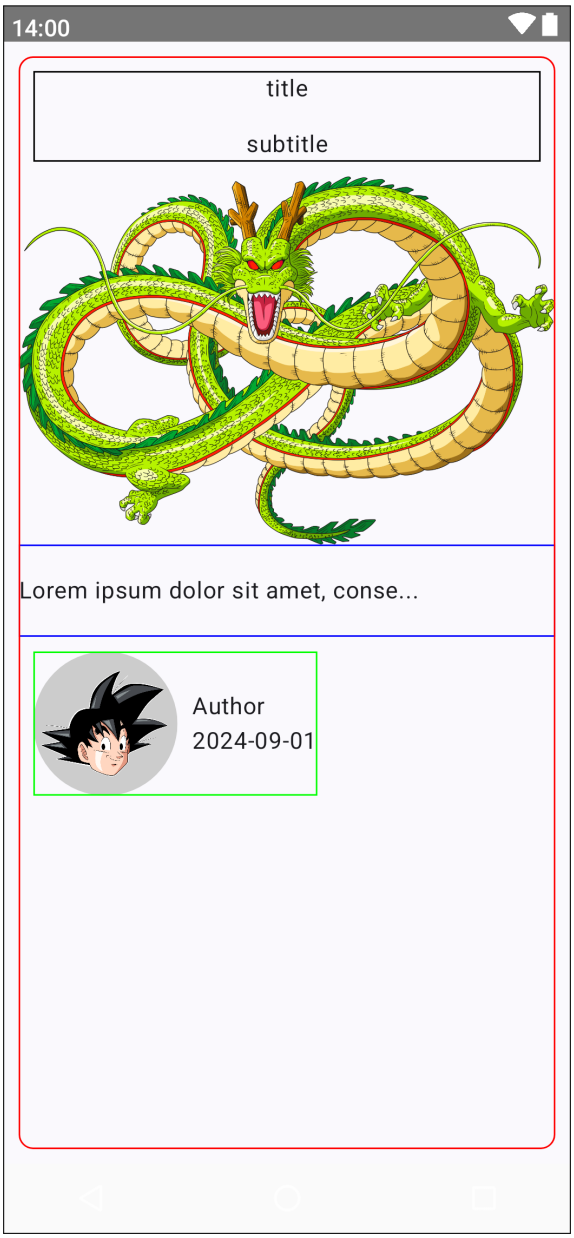
Los modificadores se pueden añadir a nivel de bloque o a nivel de elemento, por lo que habrá que tener cuidado con el tipo de sentencias que utilizamos, ya que algunas prevalecen sobre otras por tener mayor

prioridad.

Añadiendo distintos modificadores, el código quedaría:

Añadidos distintos modificadores

```
@Composable
fun Post() {
    Column (
        modifier = Modifier // FALTA CÓDIGO
    ) {
        Column (
            horizontalAlignment = Alignment.CenterHorizontally,
            modifier = Modifier
                .padding(10.dp)
                .fillMaxWidth()
                // FALTA CÓDIGO
        ) {
            Text(
                text = "title",
                modifier = Modifier.padding(bottom = 15.dp)
            )
            Text("subtitle")
        }
        Image(
            painter = painterResource(R.drawable.dragon),
            contentDescription = null
        )
        Text(
            text = "Lorem ipsum dolor sit amet, conse...",
            modifier = Modifier // FALTA CÓDIGO
        )
        Row (
            verticalAlignment = Alignment.CenterVertically,
            modifier = Modifier // FALTA CÓDIGO
        ){
            Column {
                Image(
                    painter = painterResource(R.drawable.goku),
                    contentDescription = null,
                    modifier = Modifier
                        .clip(CircleShape)
                        .background(color = Color.LightGray)
                        .size(100.dp)
                        .padding(5.dp)
                )
            }
            Column (
                Modifier.padding(start = 10.dp)
            ) {
                Text("Author")
                Text("2024-09-01")
            }
        }
    }
}
```



Se han añadido distintos modificadores tanto a nivel de columna/fila como a nivel de componente final. Tal como se puede ver, el código es bastante auto-explicativo. El código de ejemplo no está completo, se han añadido comentarios donde falta código para dibujar los bordes en algunos modificadores. **Los bordes facilitan ver la disposición de cada conjunto de elementos**

Ejercicio

Completa el código anterior añadiendo los modificadores de los bordes para que se asemeje a la imagen mostrada.

Ejercicio

Modifica el código anterior para que el contenido de los textos sean recibidos como parámetros.

3.3.1. Componentes "vagos"/lazy

En el ejemplo que estamos viendo estamos haciendo uso de los sistemas de composición *Column* y *Row* para la disposición/colocación de los distintos elementos. Nos permiten realizar la colocación de los elementos según nos interesa.

Cuando tenemos pocos elementos, o un número finito de hijos es correcto hacer uso de estos sistemas. El problema surge cuando tenemos un número no determinado de hijos como puede ser una lista de elementos. En estos casos haremos uso de ***LazyColumn*** (o ***LazyRow*** para filas), ya que sólo renderizará los elementos que están en pantalla, siendo más eficiente en estos casos.

3.3.2. Modificadores reusables

En caso de que hagamos uso de modificadores comunes en varios componentes, podemos crear una variable, y asignarla a los distintos componentes.

3.4. Vista vertical y horizontal

Si giramos el dispositivo con el ejemplo que hemos creado hasta ahora, veremos que la disposición de los elementos se mantiene de manera correcta, salvo por el hecho de que **no tenemos *scroll***. El componente *Column* no tiene scroll por defecto, por lo que deberemos añadirlo al modificador principal:

Añadidos distintos modificadores

```
@Composable
fun Post() {
    Column (
        modifier = Modifier // FALTA CÓDIGO
            .verticalScroll(rememberScrollState())
    ) {
        //...
    }
}
```

De esta manera, en la vista horizontal nuestra aplicación tendrá *scroll* al haberle dado esa característica a la columna principal.

Por otro lado, si queremos diferenciar cómo generar la composición de la vista dependiendo de si el dispositivo está en vertical u horizontal, podemos hacer lo siguiente:

Composición distinta en vertical u horizontal

```
// Cogemos la configuración actual
val configuration = LocalConfiguration.current

// Hacemos una composición diferente según estado
when (configuration.orientation) {
    Configuration.ORIENTATION_LANDSCAPE -> {
        LandscapeScreenComposition()
    }
    else -> {
        VerticalScreenComposition()
    }
}
```

3.5. Crear temas personalizados

A la hora de crear nuestra aplicación puede ser interesante tener un "tema" (en inglés *theme*) personalizado. Estos "temas" suelen hacer referencia a los colores de la aplicación, la tipografía, … Y este tema será diferente dependiendo de si tenemos el dispositivo en modo "claro" u "oscuro".

Información

La personalización de las aplicaciones se hace a través del "tema"/themes, que normalmente tendrá colores corporativos, tipos de letras elegidos, tamaños elegidos…

Al crear nuestro proyecto veremos que existe un directorio llamado `ui/theme` dentro de la misma ruta donde se encuentra el fichero `MainActivity.kt`. En ese directorio nos encontraremos con tres ficheros, que es recomendable abrir para ver su contenido:

- **Color.kt**: Contiene unas variables con los colores que van a ser utilizados en la aplicación. Los colores se pueden definir de distintas maneras y pueden tener un rango "Alpha" que es la opacidad. Podemos generar nuestro propio sistema de colores y existen distintos roles que podemos utilizar en nuestra aplicación.

¡Cuidado!

Crear un buen sistema de colores puede distinguir a nuestra marca/empresa de las otras.

- **Type.kt**: En este fichero se hace referencia a las tipografías, la fuente de letra a utilizar, tipo, tamaño, … Por defecto existen tres escalas ("**Large**", "**Medium**" y "**Small**") para los siguientes roles:
    - **Display**: El texto más grande de la pantalla, reservado para textos cortos o numéricos. Funcionan mejor en pantallas grandes. Se suelen usar fuentes "expresivas", de tipo manuscritas.

- **Headline**: Adecuados para textos breves en pantallas más pequeñas. Útiles para marcar pasajes de un texto o partes importantes de un contenido.
- **Title**: Se deben usar para textos de énfasis medio que sean relativamente cortos.
- **Body**: Se usa para textos largos. Hay que tratar de evitar fuentes muy expresivas o decorativas, porque pueden ser más difíciles de leer.
- **Label**: Se utiliza para el texto dentro de componentes. Por ejemplo, los botones usan el estilo ***LabelLarge***.

Se puede ver la explicación de cada tipo de fuente en la explicación de los estilos y un ejemplo de distintas tipografías en el siguiente enlace a un proyecto de ejemplo en Github.

- **Theme.kt**: Es el fichero principal del tema, y en el que se conjuga y utiliza lo especificado en los ficheros nombrados anteriormente. Si miramos el fichero de un proyecto recién creado, contiene lo siguiente:

Fichero Theme.kt

```
package com.example.jetpack.ui.theme
//...
private val DarkColorScheme = darkColorScheme(
    primary = Purple80,
    secondary = PurpleGrey80,
    tertiary = Pink80
)

private val LightColorScheme = lightColorScheme(
    primary = Purple40,
    secondary = PurpleGrey40,
    tertiary = Pink40
)

@Composable
fun JetpackTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    // Dynamic color is available on Android 12+
    dynamicColor: Boolean = true,
    content: @Composable () -> Unit
) {
    val colorScheme = when {
        dynamicColor && Build.VERSION.SDK_INT >= Build.VERSION_CODES.S -> {
            val context = LocalContext.current
            if (darkTheme)
                dynamicDarkColorScheme(context)
            else
                dynamicLightColorScheme(context)
```

```
        }
        darkTheme -> DarkColorScheme
        else -> LightColorScheme
    }

    MaterialTheme(
        colorScheme = colorScheme,
        typography = Typography,
        content = content
    )
}
```

- **Shapes.kt**: En un proyecto recién creado no se crea este fichero, pero en este ejemplo se puede ver su contenido. En este fichero se pueden modificar las formas que podemos utilizar en nuestra aplicación.

Ejercicio

Existen distintos tutoriales (tutorial 1 y tutorial 2) para crear temas con colores propios.

Información

Material Theme Builder nos ayuda a crear nuestro tema personalizado.

3.6. Eventos en Compose

Tal como dice la documentación oficial, Compose es declarativo y, por lo tanto, la única manera de actualizarlo es llamar al mismo elemento que admite composición con argumentos nuevos. Estos argumentos son representaciones del estado de la IU. Cada vez que se actualiza un estado, se produce una ***recomposición***.

- **Composición**: Es una descripción de la IU que compila Jetpack Compose cuando ejecuta elementos de componibilidad.
- **Composición inicial**: Es la creación de una composición con la primera ejecución de elementos componibles.
- **Recomposición**: Es la nueva ejecución de los elementos de componibilidad a los fines de actualizar la composición cuando los datos cambian.

¡Cuidado!

A un elemento que admite composición se le debe informar, de manera explícita, el estado nuevo para que se actualice según corresponda.

Compose necesita saber de qué estado se debe hacer un seguimiento a fin de que, cuando reciba

una actualización, pueda programar la recomposición. Compose solo realizará la recomponsición de las funciones que deben cambiar. Para ello usaremos los tipos State y MutableState de Compose para que Compose pueda observar el estado. La función `Mutablestate` puede actualizar su `value` para actualizar el estado.

Vamos a realizar el ejemplo del contador, cuya función sería la siguiente:

Función contador

```
@Composable
fun Contador(modifier: Modifier = Modifier) {
    var contador: MutableState<Int> = remember { mutableStateOf(0) }

    Surface(modifier = Modifier.fillMaxSize()) {
        Column(
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Text("${contador.value}")
            Button(
                onClick = {
                    contador.value++
                }
            ) {
                Text("Suma!")
            }
        }
    }
}
```

La variable “contador” es de tipo “MutableState”, que se actualiza al pulsar el botón, y Compose al detectar que se ha actualizado su “value”, recompone esa parte de la vista, mostrando el nuevo valor.

Pregunta

¿Qué pasa con los datos si rotamos la pantalla?

Ejercicio

Más sobre estados y actualizaciones de vista en este ejemplo de la documentación y en este otro

4. Ciclo de vida del *Activity* en Android

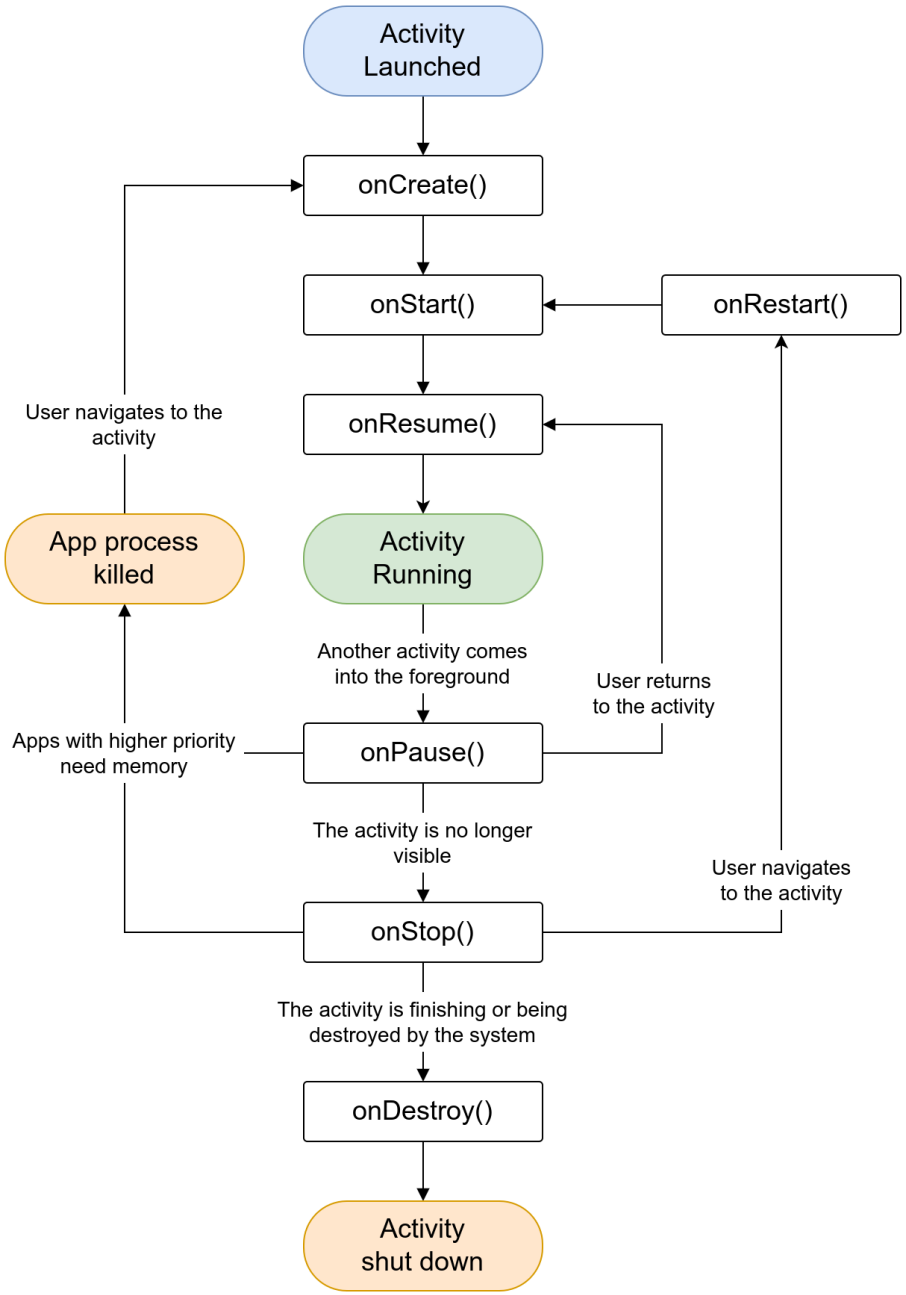
El contenido de la vista se resetea al rotar el dispositivo. Es decir, **cualquier posible contenido que hubiese en la vista que no se haya guardado se pierde**. Esto es debido a que el ***Activity*** de la vista se destruye y se vuelve a crear en el nuevo estado de la pantalla.

¡Cuidado!

Al rotar el dispositivo el *Activity* entra en un ciclo de destrucción-creación

Es importante conocer qué sucede al rotar el dispositivo, y qué sucede con la vista y con los componentes que tenemos en primer plano.

Existen distintas funciones dentro del **ciclo de vida de la actividad**, que serán llamadas por el sistema cada vez que se entre en cada uno de estos estados:



Esquema del "*Activity-lifecycle*". Fuente: Documentación oficial

En el esquema (y en la web oficial de la documentación) se puede apreciar los estados y sus transiciones.

Ahora bien, si queremos guardar el estado de variables para ser utilizadas al cambiar el estado, habrá

variación del tipo de vista que estemos utilizando, ya que con XML o con Compose es distinto. De nuevo, en la documentación sobre cómo guardar estados de la IU nos indicará qué métodos debemos usar.

4.1. Mantener los estados con XML

Una vez sabemos cómo actúa las distintas transiciones de estados de las vistas, es momento de poder guardar el estado para poder recuperar la información al crear el Activity de nuevo. Para ello se van a usar dos funciones:

- **onSaveInstanceState** : Se llamará cuando vaya a destruirse el Activity.
- **onRestoreInstanceState**: Se llamará justo después de llamar al método **onCreate**.

Siguiendo con el ejemplo previo, donde queremos guardar el número de veces que se ha pulsado el botón, deberíamos añadir algo como lo siguiente:

Código Kotlin para mantener estados

```kotlin
override fun onResume(){
    super.onResume()
    Log.d("Debug","onResume")
    val text:TextView = findViewById(R.id.textView)
    text.text = contador.toString()
}

override fun onSaveInstanceState(outState: Bundle) {
    // Save the user's current game state.
    outState?.run {
        putInt("contador", contador)
    }
    // Always call the superclass so it can save the view hierarchy.
    super.onSaveInstanceState(outState)
}

override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    // Always call the superclass so it can restore the view hierarchy.
    super.onRestoreInstanceState(savedInstanceState)
    // Restore state members from saved instance.
    savedInstanceState?.run {
        contador = savedInstanceState.getInt("contador")
    }
}
```

Ejercicio

Añade al **MainActivity.kt** las funciones necesarias para ver todas las transiciones al rotar la pantalla. Que las funciones muestren en el **Logcat** un texto que indique qué se ha ejecutado.

4.2. Mantener los estados con Compose

El ciclo de vida de los elementos componibles es diferente en Jetpack Compose, ya que la *recomposición* se realiza a nivel de componente en lugar de a vista completa.

Para guardar el estado de la variable entre recomposiciones, haremos uso del tipo "rememberSaveable". Esto nos permitirá rotar el dispositivo y mantener el valor de la variable:

Mantener el estado

```
@Composable
fun Contador(modifier: Modifier = Modifier) {
    var contador: MutableState<Int> = rememberSaveable { mutableStateOf(0) }
    // ...
}
```

Al hacer uso de "rememberSaveable" nos permite guardar el estado durante la recomposición, pero dado que el ciclo de vida de la actividad puede contener otros estados, no siempre será funcional.

¡Cuidado!

Usar "rememberSaveable" no guarda el estado de la recreación de la actividad.

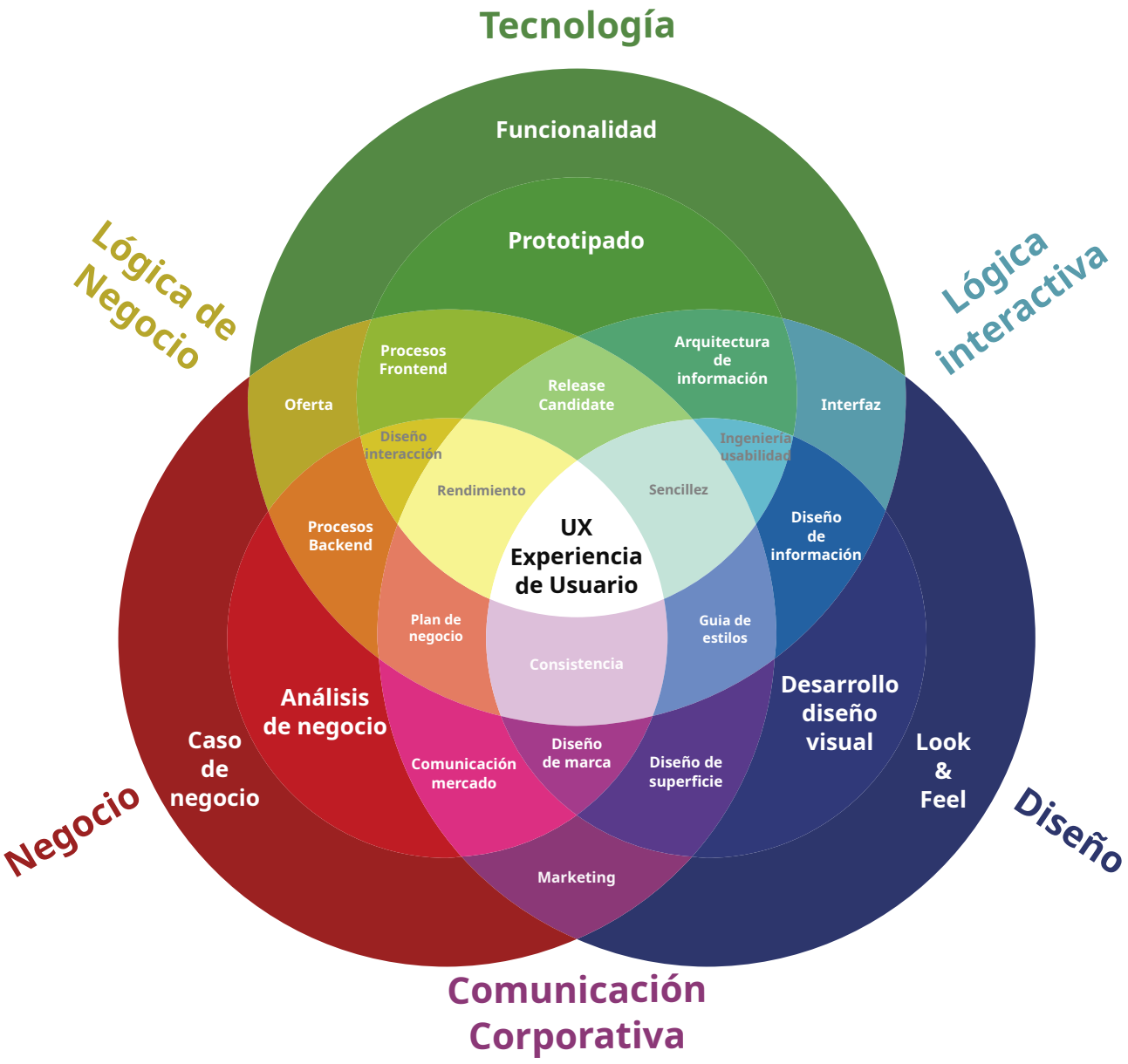
Para hacer uso más complejo de los estados, en cómo guardar estados de la IU y en estado y Jetpack Compose hay más documentación sobre ello.

III

Desarrollo de aplicaciones

1. Introducción

Durante el desarrollo de aplicaciones hay que tener en cuenta ciertas cuestiones que deben ser pensadas para asegurar que la aplicación cumple con las expectativas y la funcionalidad deseadas. De no tener en cuenta algunos de estos aspectos, la aplicación resultante puede resultar difícil de usar, no ajustarse a los objetivos previstos, contener fallos de usabilidad… Es más, todos los aspectos de una aplicación están relacionados entre sí, tal como se puede ver en este gráfico.

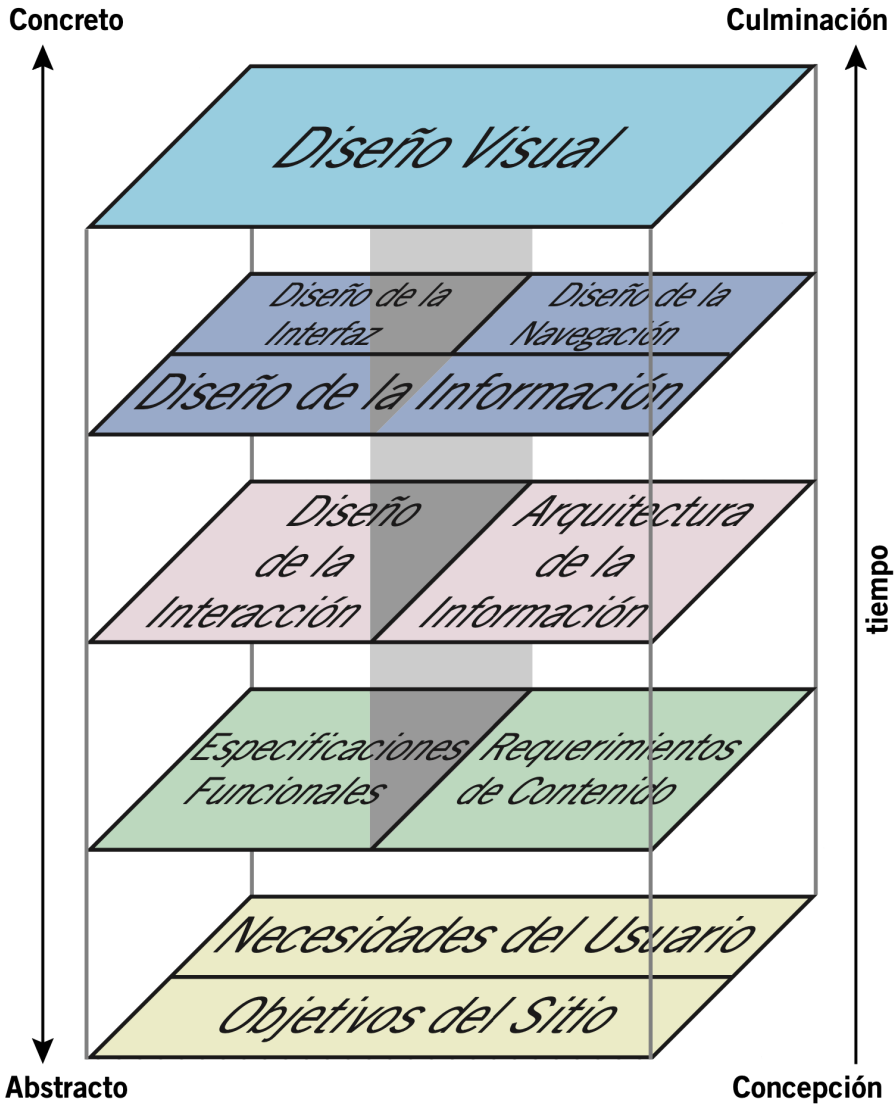


Aspectro de la experiencia de usuario. Referencia.

Si simplificamos el aspectro a tener en cuenta y nos centramos sólo en las necesidades que puede tener un proyecto, los elementos a tener en cuenta para la experiencia de usuario también se simplifican, **aunque mantienen la importancia**.

En este aspecto, Jesse James Garrett divide los elementos que tienen relación con la experiencia de usuario

en varios elementos separados en cinco capas. Aunque se hacía referencia a los sistemas web, también son válidos en aplicaciones de escritorio y móvil.



Elementos de UX según Jesse James Garrett. Fuente traducida

2. Definiendo conceptos

Durante el desarrollo de una aplicación, en lo que se refiere a su uso, existen distinta terminología, que en algunos casos, se utilizan como sinónimos, o entre-mezclados, lo que dificulta su comprensión.

2.1. Diseño de interacción

Tal como nos dice la wikipedia, el **diseño de interacción** (abreviado como **IxD**), está definido como "la práctica de diseñar productos digitales interactivos, entornos, sistemas, y servicios". Esta definición no sólo afecta a entornos digitales, ya que también es aplicable al mundo real.

Error en el diseño

Podríamos resumirlo como:

- Cuáles son las acciones que el usuario tiene que realizar para llevar a cabo su objetivo al utilizar la aplicación.
- Cómo debe realizar dichas acciones.

Es importante diseñar sistemas que sean efectivos, eficientes y sencillos a la hora de utilizarlos, ya que por el contrario, por muy buena que sea nuestra aplicación, los usuarios pueden decidir no utilizarla.

¡Cuidado!

Si nuestra aplicación no es sencilla de utilizar, los usuarios pueden dejar de usarla.

El objetivo debe ser minimizar los errores que los usuarios pueden realizar, incrementar la satisfacción del usuario, disminuir la frustración y, en definitiva, hacer más productivas las tareas que rodean a las personas y la aplicación.

2.2. Usabilidad

Usabilidad es un neologismo (del inglés *usability*, facilidad de uso) que se refiere a la facilidad con que las personas pueden utilizar una herramienta u objeto. En el mundo de la informática es aplicable a una aplicación, página web, máquina, proceso…

La **usabilidad** también se refiere al estudio de esas herramientas, normalmente previo o durante al desarrollo, para tratar de determinar cuál será el **grado de usabilidad** de las mismas. Para ello se realizarán **pruebas empíricas y relativas**:

- **Empírica** porque no se basa en opiniones o sensaciones, sino en pruebas de usabilidad realizadas en laboratorio u observadas mediante trabajo de campo (por ejemplo, con usuarios).

- **Relativa** porque el resultado no es ni bueno ni malo, sino que depende de las metas planteadas o de una comparación con otros sistemas similares.

Información

Jakob Nielsen definió la usabilidad como el atributo de calidad que mide lo fáciles que son de usar las interfaces Web.

Jakob Nielsen, en su libro *Usability Engineering (1993)*, define los cinco componentes de calidad de sus "Objetivos de Usabilidad":

- **Capacidad de aprendizaje**: ¿Cómo de fácil es para los usuarios realizar tareas básicas la primera vez que conocen el diseño?
- **Eficiencia**: Una vez que los usuarios han aprendido el diseño, ¿con qué rapidez pueden realizar tareas?
- **Memorabilidad**: Cuando los usuarios vuelven al diseño después de un período sin usarlo, ¿con qué facilidad pueden recuperar la competencia?
- **Errores**: ¿Cuántos errores cometen los usuarios, cómo de graves son estos errores y con qué facilidad pueden recuperarse de ellos?
- **Satisfacción**: ¿Cómo de agradable es utilizar el diseño?

Información

Existe la norma ISO 9241 enfocada a la calidad en usabilidad y ergonomía del hardware y software.

2.3. Experiencia de usuario

La experiencia del usuario (**UX**) es el conjunto de factores y elementos relativos a la interacción del usuario con un entorno o dispositivo concretos, dando como resultado una percepción positiva o negativa de dicho servicio, producto o dispositivo.

La experiencia de usuario es un concepto subjetivo, ya que junto con la interacción del usuario también pueden ir acompañadas de aspectos relativos como las emociones, los sentimientos y la transmisión de la marca…

Inicialmente la experiencia de usuario se le empezó a dar importancia en la informática durante el auge de internet y el desarrollo páginas web.

Para diseñar una buena experiencia de usuario, debemos partir de un diseño de interacción adecuado y de un buen diseño de interfaz.

Por otro lado, la experiencia de usuario también puede verse alterada por la predisposición del usuario y su propia experiencia: tratar de utilizar una aplicación como otra parecida,

Información

Muchas aplicaciones con objetivos similares suelen tener interfaces similares para facilitar la transición de usuarios.

2.4. Interfaz de usuario

La interfaz de usuario (en inglés *user interface*, **UI**) es la capa que se sitúa entre el usuario y el propio producto (da igual que sea una aplicación o algo físico).

En esta interfaz de usuario aparecerán todos los elementos con los que el usuario interactuará de manera directa. Estos elementos generarán eventos o realizarán algún tipo de actividad en el producto, que normalmente generará una salida/respuesta.

2.4.1. Procesos de diseño de un interfaz de usuario

El diseño de la interfaz de usuario es una disciplina asociada al diseño industrial y se enfoca en maximizar la usabilidad y la experiencia de usuario.

Antes de realizar el diseño del interfaz de usuario debemos tener claro cómo queremos que sea la interacción entre la persona que va a utilizar la aplicación y la propia aplicación.

¡Cuidado!

Para diseñar el interfaz, debemos tener clara la interacción humano-aplicación.

Para el diseño de la interfaz deberemos tener en cuenta, al menos, los siguientes puntos:

- Elaborar una **lista de los elementos funcionales** requeridos por el sistema para que cumpla los objetivos del proyecto. Esto se realiza teniendo claro los **requisitos** del proyecto.
- **Analizar los usuarios potenciales de la aplicación y las tareas que van a realizar**. Normalmente se hace a través de un trabajo de campo estudiando la forma en la que realizan las tareas, entrevistando a los usuarios, … Algunas de las preguntas que debemos contestar son:
  - ¿Qué querría el usuario que haga el sistema?
  - ¿Cómo encajaría el sistema en el flujo de trabajo o las actividades diarias?
  - ¿Cuán competente es el usuario técnicamente y qué sistemas parecidos ya utiliza?
  - ¿Qué estilos de aspecto y comportamiento son los preferidos del usuario?
- Es importante tener clara cómo es la **arquitectura de la información**, que es el estudio, análisis, organización, disposición, estructuración y presentación de la información.
- A la hora de diseñar interfaces, **es útil hacer uso de sistemas de prototipado** utilizando herramientas específicas para ello. En algunos casos, incluso pueden tener una funcionalidad básica.

- Ya se ha mencionado previamente, pero es importante recalcar realizar **pruebas de usabilidad**, en las que el usuario pueda expresar lo que piensa mientras utiliza la aplicación.

2.4.2. Principios y requisitos de diseño de una interfaz de usuario

Las características dinámicas de un sistema se describen en términos de requisitos diálogo

La norma ISO 9241 establece una serie de conceptos y elementos básicos de ergonomía que suponen un punto de partida para facilitar el diálogo entre los sistemas y las personas que usan dichos sistemas, con **definiciones de alto nivel**, **aplicaciones ilustrativas** y **ejemplos de los principios definidos**. Los principios aplicables representan los aspectos dinámicos de la interfaz y pueden considerarse, de forma general, como la "sensación" que produce la interfaz.

Tal como nos dice la wikipedia, los siete principios son los siguientes:

- **Adecuación a la tarea**: el diálogo es adecuado a la tarea cuando asiste al usuario en la compleción eficaz y eficiente de la tarea.
- **Carácter autodescriptivo**: el diálogo es autodescriptivo cuando cada paso del diálogo es inmediatamente comprensible ya sea mediante la información devuelta por el propio sistema o por una explicación a solicitud del - usuario.
- C**onformidad con las expectativas del usuario**: el diálogo es conforme con las expectativas del usuario cuando es consistente y se ajusta a las características del usuario, tales como conocimiento de la tarea, - educación, experiencia, y otros convenios comúnmente aceptados.
- **Adecuación al aprendizaje**: el diálogo es adecuado al aprendizaje cuando ofrece soporte y guía para que el usuario aprenda a utilizar el sistema.
- **Controlabilidad**: el diálogo es controlable cuando el usuario es capaz de iniciar y controlar la dirección y ritmo de la interacción hasta el punto en el que la tarea ha sido completada.
- Tolerancia a errores: el diálogo es tolerante a errores si, con independencia de que haya errores de la entrada, el resultado pretendido puede ser alcanzado sin acción necesaria por parte del usuario, o con una acción - mínima.
- Personalizable: el diálogo es personalizable cuando la interfaz de software puede ser modificada para ajustarse a las necesidades de la tarea, preferencias individuales, y habilidades del usuario.

En cuanto a la **presentación de la información**, cómo se organiza en la aplicación, la visualización de los objetos y la codificación de la información, también se tiene que tener en cuenta varios puntos:

- **Claridad**: el contenido de la información es presentado de forma rápida y precisa.
- **Discriminabilidad**: la información visualizada puede ser distinguida de forma precisa.
- **Concisión**: los usuarios no son sobrecargados con información irrelevante.
- **Consistencia**: el diseño es único y conforme a las expectativas del usuario.

- **Detectabilidad**: la atención del usuario es dirigida hacia la información necesaria.
- **Legibilidad**: la información es fácil de leer.
- **Comprensibilidad**: el significado es claramente inteligible, no ambiguo, interpretable y reconocible.

3. Toma de requisitos del cliente

Antes de comenzar con la programación de una aplicación, debemos tener claro cuáles son los requisitos del cliente cuál es el alcance del proyecto, qué se espera del resultado.

Estos requisitos se suelen obtener mediante entrevistas con el cliente para recabar toda la información posible, para que de esta manera queden claras las funcionalidades que debe tener la aplicación.

En caso de que algún requisito no sea aclarado al inicio del proyecto, puede desencadenar en una cadena de tomar malas decisiones que repercutirá en el resultado final. El problema es que en algunos casos, esas malas decisiones pueden incluirse en el *core* de la aplicación y limitarla a futuro.

¡Cuidado!

Una mala toma de requisitos puede hacer que la aplicación se desarrolle de manera incorrecta y limitarla a futuro.

3.1. Obtención de los requisitos

Tal como se ha dicho previamente, la obtención de requisitos se realiza al inicio del proyecto, antes de realizar ningún tipo de programación. Las maneras más habituales de obtener los requisitos son:

- Entrevistas con el cliente.
- Entrevistas con los usuarios que van a usar la aplicación.
- Conocer y analizar el estado actual (si lo tienen) de la metodología de trabajo. Ya que con la finalización del proyecto, esta metodología se deberá adaptar.

También es importante que el analista y jefe de proyecto se pongan en la piel del cliente, para poder hacer las preguntas oportunas y poder entender qué se quiere realizar.

3.2. Análisis de requisitos

Una vez obtenida la información por parte del cliente, se debe realizar un análisis de los requisitos, de esta manera, pueden surgir dudas, y en ese caso, volver al paso anterior.

Diferenciar los tipos de requisitos nos ayudará a entender el alcance de cada uno de ellos para así poder gestionar distintos apartados de la aplicación.

A la hora de analizar los requisitos, se suele crear una tabla donde se indicará:

- **Tipo** de requisito.

- **Categoría**, para poder separarlos entre los distintos tipos.
- **Prioridad**, para poder ordenarlos en base a lo prioritoario que son, y cuándo se deben abordar.
- **Dependencias**, ya que hay requisitos que pueden depender de otros. Por lo tanto, antes de poder realizar el requisito, deben terminarse las dependencias.
- **Título** descriptivo que resuma el requisito.
- **Descripción** del requisito, donde habrá que dar toda la información que sea necesaria para posteriormente poder desarrollarla.
- **Razón** del requisito, para poder entender mejor por qué debe realizarse.

A la hora de escribir los requisitos, podemos crear una tabla para cada uno de ellos de la siguiente forma:

<table>
<tr><th>ID
1</th><th>Tipo
Funcional</th><th>Categoría
Transaccional</th><th>Prioridad
Muy Alta</th><th>Dependencias</th></tr>
<tr><td colspan="5">Registro de usuarios</td></tr>
<tr><td colspan="5">Descripción:
La plataforma permitirá el registro del usuario solicitando los siguientes datos:
• E-mail
• Contraseña
• Nombre y apellidos (opcional)
• Fecha de nacimiento
• Género (opcional)
• País (opcional)</td></tr>
<tr><td colspan="5">Razón:
Para poder registrar recetas el usuario debe estar registrado en el sistema. Algunos datos son opcionales. La fecha de nacimiento es obligatoria ya que los usuarios deben ser mayores de edad.</td></tr>
</table>

En este caso, en el apartado “Dependencias” no se ha añadido nada ya que es el primer requisito de la aplicación.

3.2.1. Requisitos de negocio

Los requisitos de negocio suelen ser los menos técnicos, y se puede resumir en **qué quiere conseguir la empresa tras finalizar el proyecto**.

Algunos ejemplos:

- Crear un portal web de recetas, para que los usuarios puedan subir sus recetas, valorarlas y compartirlas.
- Crear una aplicación que mejore la productividad de la gestión de productos de la empresa, para tener un inventariado digital, que genere avisos antes de que se agoten.

- Crear una red social de gente que le guste la escalada, donde poder compartir rutas, valorar dificultades, hacer quedadas para escalar.

Tal como se puede ver, es una idea general, pero que ya nos da una idea de otros requisitos que tendremos que tener en cuenta durante el desarrollo.

3.2.2. Requisitos funcionales

Los requisitos funcionales son las declaraciones de los servicios que debe proporcionar la aplicación. En definitiva, definir cómo se va a comportar el sistema en distintas situaciones a lo largo de la aplicación.

Los requisitos funcionales se pueden dividir en:

- Requisitos transaccionales
- Requisitos de datos
- Requisitos de interfaz o de presentación
- Requisitos de personalización

3.2.2.1. Requisitos transaccionales

Los requisitos transaccionales los podemos resumir como las funcionalidades que tendrá la aplicación, así como las tareas que el usuario podrá realizar con los datos contenidos en la aplicación.

En base a estos requisitos transaccionales y estas funcionalidades, veremos cómo posteriormente surgirán nuevos requisitos (como pueden ser los requisitos de datos y requisitos de interfaz).

Un ejemplo de un requisito transaccional es el visto anteriormente, como **registro de usuarios**, y otro puede ser el **realizar login de usuario**.

3.2.2.2. Requisitos de datos

Los requisitos de datos, como su propio nombre indica, tiene que ver con los datos que la aplicación va a utilizar. Es importante entender que los requisitos funcionales que salen de los puntos anteriores requerirán a su vez de los datos que la aplicación va a utilizar.

Se debe entender el análisis realizado para entender qué datos se van a utilizar, para posteriormente realizar el esquema Entidad-Relación de la base de datos.

Información

Partiendo de los requisitos de datos podemos generar la base de datos de la aplicación.

Ejemplos:

- **Datos de usuario**: Qué datos se pedirán al usuario al realizar el registro.
- **Datos de contenido**: Si nuestra aplicación es de recetas de cocina, qué datos se van a pedir al añadir una nueva receta (nombre, ingredientes, dificultad, procedimiento de elaboración,…).

3.2.2.3. Requisitos de interfaz/presentación

A la hora de representar la información y de hacer uso de la aplicación, el usuario deberá interactuar con el sistema y esto se hace mediante distintos elementos del interfaz. Es por eso que es necesario tener claro cuáles son los requisitos de interfaz, y partiendo de estos, generar un mapa de navegación

Ejemplos:

- **Registro de usuario**: Panel donde el usuario se podrá registrar.
- **Login de usuario**: Interfaz donde se puede realizar login.
- **Interfaz de usuario**: Una vez un usuario se ha logueado, qué se le mostrará al usuario en su panel principal.

3.2.2.4. Requisitos de personalización

Estos requisitos pueden ser opcionales, ya que no todas las aplicaciones permitirán un sistema de personalización para los usuarios. En caso de poder realizar algún tipo de personalización, en este apartado se detallarían.

Ejemplos:

- Poder elegir idioma de la aplicación.
- El cambio de idioma se realizará de manera automática.

3.2.3. Requisitos no funcionales

Los requisitos no funcionales son aquellos que no se refieren directamente a las funciones específicas que proporciona el sistema. Estos requisitos no funcionales también los podemos diferenciar en distintos apartados como vamos a ver a continuación.

Información

Los requisitos no funcionales suelen ser aspectos técnicos.

3.2.3.1. Requisitos de producto

Estos son los requerimientos que especifican el comportamiento del producto y que pueden verse asociados a la eficiencia, fiabilidad, disponibilidad… Normalmente son detalles técnicos que el cliente no tiene por qué conocer, y que por tanto debemos abordar desde la parte técnica.

Hay que tener en cuenta, que en este caso los requisitos serán muy distintos dependiendo del tipo de producto/proyecto que estemos realizando, ya que no será igual una aplicación web o una aplicación móvil.

Ejemplos:

- La aplicación web debe estar disponible 24x7 los 365 días del año.

- El despliegue de nuevas versiones se realizará mediante un sistema que pueda revertir los cambios en caso de error.
- La aplicación móvil se testeará internamente antes de enviarla a la *store* correspondiente.

3.2.3.2. Requisitos organizacionales

Dentro de los requisitos no funcionales también podemos distinguir los denominados como organizacionales, entre los que podríamos destacar, entre otros:

- **Requisitos de entrega**: Cuándo se debe realizar la entrega de los distintos *sprints* de la aplicación y de la versión final.
- **Requisitos de implementación**: Lenguaje de programación a utilizar, *framework* o versiones utilizadas, sistema de *debug*, cómo se realiza el seguimiento del proyecto, sistema de control de versiones a utilizar…
- **Uso de estándares**: Si es un lenguaje de programación compilado, qué compilador y versión se va a utilizar. Qué sistema se va a usar a la hora de crear el nombre de las variables (camelCase, snake_case, …). Si es web, uso de HTML5+CSS3…

3.2.4. Requisitos del sistema

Para finalizar, los requisitos del sistema hacen referencia a dónde se va a ejecutar el proyecto, y cuáles son las garantías que debemos abordar para que funcione de manera correcta.

De nuevo, existirán diferencias en caso de que el proyecto sea una aplicación web o una aplicación de móvil.

Ejemplos:

- La aplicación deberá funcionar en sistemas Android con API 30 o superior
- La aplicación móvil no necesitará ningún permiso para ser usada.
- La aplicación web será desplegada en AWS en distintas zonas de computación.

4. Diseño de modelos conceptuales

Tras la toma de requisitos es el momento de analizarlos y realizar los correspondientes diseños conceptuales. Estos modelos son la base para posteriormente comenzar con la programación.

Tal como nos dice Wikipedia, **un modelo conceptual es una representación de un sistema**, hecho de la composición de conceptos que se utilizan para ayudar a las personas a conocer, comprender o simular un tema que representa el modelo, incluye las entidades importantes y las relaciones entre ellos.

El objetivo principal de un modelo conceptual es transmitir los principios fundamentales y la funcionalidad básica del sistema que representa. Además, debe desarrollarse de tal manera que proporcione una interpretación del sistema fácilmente comprensible para los usuarios del modelo.

Los modelos conceptuales tienen como objetivos fundamentales:

1. Mejorar la comprensión individual del sistema representativo.
2. Facilitar la transmisión eficiente de los detalles del sistema entre las partes interesadas.
3. Proporcionar un punto de referencia para los diseñadores de sistemas para extraer las especificaciones del sistema.
4. Documentar el sistema para futuras referencias y proporcione un medio para la colaboración.

Entre los tipos de modelos que nos podemos encontrar, podemos destacar:

- Diagramas de estructura
  - Diagramas de clases
  - Diagrama de datos
  - Diagramas de componentes
  - Diagramas de despliegue
- Diagramas de comportamiento e interacción
  - Diagramas de actividades
  - Diagramas de casos de uso
  - Diagramas de estado
  - Diagramas de secuencia

A continuación se detallan algunos de ellos.

4.1. Diseño conceptual de datos

Para la realización del diseño conceptual de datos, lo más habitual suele ser utilizar el conocido "**modelo Entidad-Relación**" que es utilizado para posteriormente realizar el diseño lógico, y finalmente el diseño físico de base de datos. Esto siempre que necesitemos un sistema de base de datos relacional.

El modelo de datos suele ser la parte fundamental de muchos tipos de aplicaciones, por lo que la toma de requisitos debe de haber sido correcta, y la creación del diseño del modelo de datos también.

Información

El diseño conceptual del modelo de datos, junto con los requisitos de datos, es una parte fundamental de muchas aplicaciones.

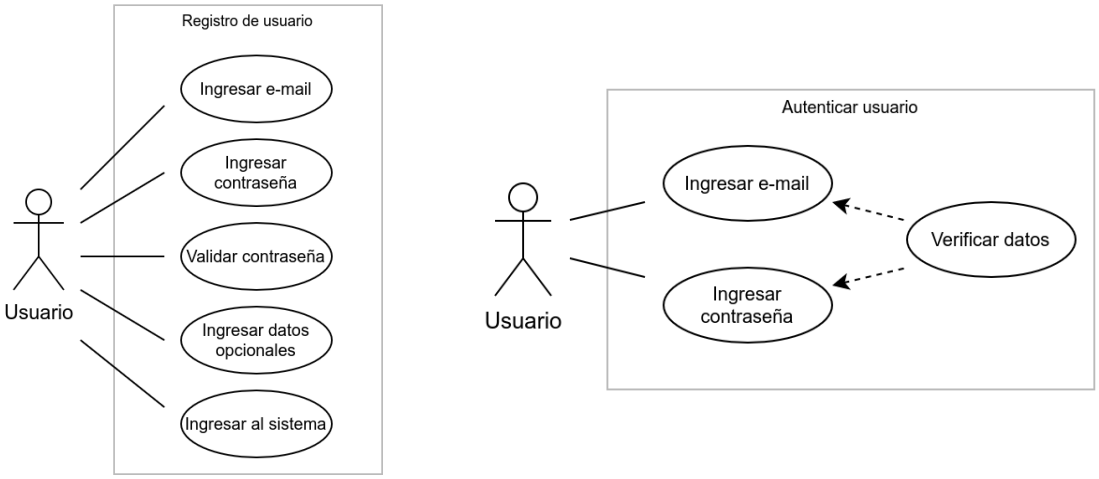
4.2. Diagramas de caso de uso

Los diagramas de caso de uso **describen una funcionalidad**, que combina la interacción entre un usuario y un sistema, formando una secuencia de eventos.

La descripción del caso de uso se centra en qué se va a realizar, pero no en la manera en la que se va a hacer a nivel técnico. Debe usarse expresiones precisas, para evitar confusión, por lo que se busca sencillez

y claridad.

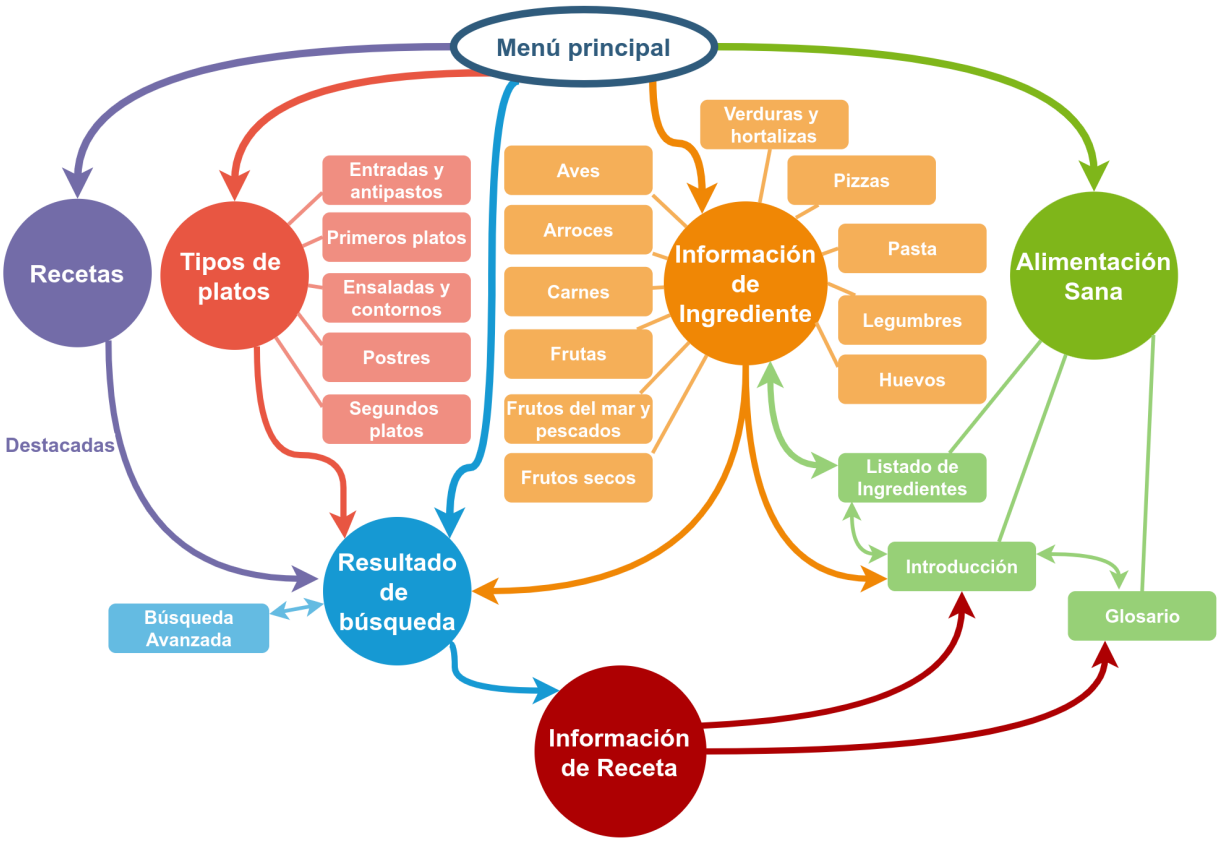
Por ejemplo, a continuación dos ejemplos de cómo se podría representar el registro de un usuario y la autenticación en un sistema, haciendo uso de diagramas de caso de uso:



Tal como se puede ver, son diagramas abstractos que muestran de forma sencilla la acción a realizar.

4.3. Mapa de navegación

Teniendo en cuenta los distintos diagramas de comportamiento e interacción que hayamos obtenido, el siguiente paso es el de realizar la topología de la aplicación y los caminos entre las distintas funcionalidades que tendrá. El resultado obtenido será el del **mapa de navegación de la aplicación**.



Mapa de navegación de una web de recetas

El mapa de navegación de una aplicación, o una web, nos debe proporcionar una descripción general, y condensada, de los enlaces entre las principales áreas de contenido.

4.4. Prototipos

A medida que los pasos anteriores se van realizando, en paralelo se puede ir realizando el análisis de cómo queremos que sea la aplicación, en lo que se refiere al aspecto visual y el diseño del interfaz.

En los primeros pasos, se pueden realizar **bocetos en papel** o en aplicaciones, sin demasiado detalle, para acompañar a los distintos diseños que se han visto previamente. Estos bocetos pueden empezar a formar una idea genérica de cómo será la aplicación.

Para realizar modelos más concretos, y más cercanos a la realidad, se hace uso de **sistemas de prototipos**, que son un modelo (representación, demostración o simulación) fácilmente ampliable y modificable. Las características que suelen tener:

- Suelen contar con interfaz de usuario.
- Pueden tener funcionalidad de entrada y salida, así como de navegación.
- Existen distintas herramientas con las que realizarlos.
- Pueden tener distinto nivel de detalle.
- El cliente es capaz de entender lo que ve, por lo que es una manera de obtener *feedback* de manera rápida.
- Nos puede mostrar aspectos imprevistos no analizados en etapas anteriores. Por lo tanto, ayuda a mejorar y concretar las especificaciones de requisitos.