

Ciclo de desarrollo de una aplicación Rich Media

Daniel de Fuenmayor López
Marcos González Sancho

PID_00192300



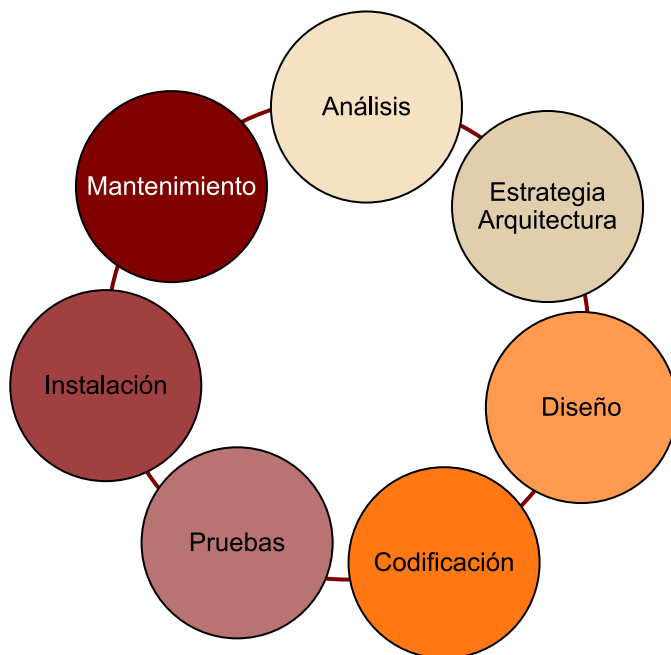
Los textos e imágenes publicados en esta obra están sujetos –excepto que se indique lo contrario– a una licencia de Reconocimiento-Compartir igual (BY-SA) v.3.0 España de Creative Commons. Se puede modificar la obra, reproducirla, distribuirla o comunicarla públicamente siempre que se cite el autor y la fuente (FUOC. Fundació per a la Universitat Oberta de Catalunya), y siempre que la obra derivada quede sujeta a la misma licencia que el material original. La licencia completa se puede consultar en: <http://creativecommons.org/licenses/by-sa/3.0/es/legalcode.ca>

Índice

1. Visión global.....	5
1.1. Enfoque en cascada	5
1.2. Enfoque evolutivo iterativo	6
1.3. Adaptación para la creación de Aplicaciones Rich Media	7
2. Planificación y estrategia.....	9
2.1. Análisis de requerimientos y funcional	9
2.2. Definición de la estrategia (diseño de software)	10
2.2.1. Definición de la/s plataforma/s destino	11
2.2.2. Arquitectura/enfoque	13
2.2.3. Tecnología/s	14
2.2.4. Planificación	15
3. Diseño, usabilidad, experiencia de usuario y accesibilidad....	16
3.1. Prototipado. Definición y concepto	16
3.2. Tipos de prototipos	18
3.3. Patrones de diseño	21
3.4. Ejemplos de patrones de diseño	22
3.4.1. Interacción básica	22
3.4.2. Otras referencias	29
3.5. Usabilidad	31
3.6. Experiencia de usuario	32
3.7. Equilibrio entre usabilidad y experiencia de usuario	32
3.8. Accesibilidad	33
3.8.1. Ideas preconcebidas sobre Flash	33
3.8.2. La accesibilidad como parte integrante del diseño de aplicaciones Rich Media	34
3.8.3. Accesibilidad en Flash	35
4. Codificación.....	38
4.1. Prototipos de desarrollo	38
4.2. Patrones de diseño	39
5. Pruebas.....	41
6. Instalación y puesta en marcha.....	43
7. Mantenimiento.....	44

1. Visión global

Una visión general del ciclo de desarrollo de una ARM nos ayudará a tener claros algunos conceptos fundamentales en cualquier proceso de desarrollo de software. La siguiente imagen muestra un ciclo habitual del desarrollo de una ARM:



Ciclo de desarrollo de una aplicación

En la imagen, se puede observar que las principales fases que definen el proceso son las siguientes:

- Análisis de requerimientos.
- Definición de la estrategia/arquitectura.
- Diseño.
- Codificación.
- Pruebas.
- Instalación y puesta en marcha.
- Mantenimiento.

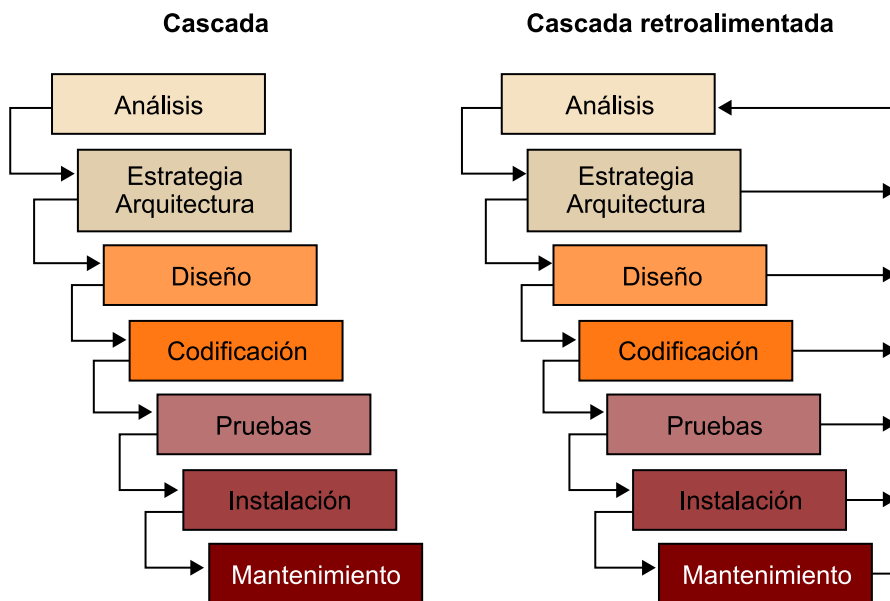
Se le pueden dar **diferentes enfoques** a este modelo de trabajo. Veamos algunos de los más empleados en la actualidad.

1.1. Enfoque en cascada

El enfoque en cascada (también conocido como clásico o secuencial) implica que cada fase nos lleva a la siguiente, en una dirección descendente y, en la teoría, sin marcha atrás. En la realidad, este modelo no deja de ser utópico, ya

que es imposible que en las primeras fases no quede nada abierto o sin definir correctamente, como para tener que resetear todo el proceso en el momento de encontrar un requisito sin definir o identificar.

Podemos decir que este enfoque es teóricamente el caso ideal, en el que se lleva a cabo una recogida de requerimientos tan exacta y correcta que permitiría, a partir de ese momento, completar el ciclo de desarrollo sin tener que ajustar fases anteriores.



Enfoque en cascada y en cascada retroalimentada

La variante lógica de este enfoque para adaptarlo a una realidad inevitable es lo que se conoce como **modelo en cascada retroalimentada**, que permite ir hacia atrás en las diferentes fases a partir de problemas y aspectos detectados en fases posteriores, para realizar los ajustes necesarios y retomar nuevamente las partes afectadas en las siguientes fases. Esta variante es muy empleada, aunque suele ser apta para desarrollos en los que el nivel de variabilidad del proyecto es muy bajo y en los que no se prevé una alta carga evolutiva del mismo.

1.2. Enfoque evolutivo iterativo

Este enfoque está pensado para adaptarse a desarrollos que tienen un grado de variabilidad mayor y necesitan de una evolución en el tiempo. No hay que confundirlo con el modelo en cascada retroalimentado, porque la filosofía que hay detrás del mismo es diferente.

Este modelo trabaja sobre versiones en las que se va iterando, para ir ampliando las funcionalidades de las mismas en base a los nuevos requerimientos que se generan. Es decir, la mecánica de este modelo se basa, precisamente, en la evolución y ampliación de requerimientos que se van abordando en diferentes

fases iterativas, que van ampliando las anteriores. La gran diferencia es que esta evolución no surge como una necesidad ante el hecho de encontrar un requerimiento mal especificado o identificado, sino como filosofía de trabajo.

En el modelo evolutivo iterativo se parte de unos requerimientos base (que no necesariamente tienen que estar definidos al detalle) para llegar a una versión que se valida y prueba, y de la que se obtiene un nuevo análisis de requerimientos evolucionado, que nos llevará a una siguiente versión del software para dar respuesta a esas funcionalidades con más detalle. Estos pasos se van repitiendo de forma constante realizando iteraciones sobre el producto, hasta llegar a un modelo estable (aunque raramente un software que se realiza con este modelo, llega a una fase final, sino que siempre está en constante evolución). En el fondo, el modelo evolutivo iterativo no deja de ser una repetición sistemática de un modelo en cascada retroalimentada a bajo nivel.

El desarrollo de forma iterativa permite simultanear tareas en diferentes áreas en diferentes iteraciones, es decir, mientras una parte del equipo está codificando o testeando una versión, otra parte podría estar trabajando ya sobre siguientes iteraciones en fases tempranas de las mismas.

La filosofía de las iteraciones no es entregar en cada una de ellas un producto no operativo en algunas de sus partes, sino que sea completamente funcional y terminado en los requerimientos marcados, aunque se sepa que sobre el mismo se tendrán que crear nuevos requerimientos y funcionalidades a resolver en posteriores iteraciones.

1.3. Adaptación para la creación de Aplicaciones Rich Media

En el campo de las Aplicaciones Rich Media se conjugan, habitualmente, una serie de factores que hacen que no exista un enfoque claro definido. Normalmente, el desarrollo se lleva a cabo por equipos de desarrollo muy pequeños, y el alto peso que tiene la interacción y el diseño en las mismas hace que el trabajo con el cliente pueda ser muy variable y complicado.

Ante esta situación, en cada proyecto habrá que validar si es mejor un enfoque u otro, aunque, cuando se trabaja para agencias en el área de marketing y publicidad (muy relacionado con las ARM y sector en el que se da un gran volumen de trabajo), es muy probable que los tiempos de desarrollo sean muy cortos, de modo que un enfoque evolutivo suele ser muy difícil de asimilar por cuestiones de calendario.

Ante esta situación, cada director de proyecto puede tomar la decisión que le parezca más adecuada, e incluso mezclar ambos enfoques en diferentes aspectos de una misma aplicación. Cuando el equipo de trabajo lleva tiempo junto, es muy habitual que se encuentre una metodología de trabajo que encaja bien con los perfiles y con las capacidades de cada miembro del equipo.

En cualquier caso, es fundamental siempre que la toma de requerimientos sea lo más fiable posible y que siempre se trate de asegurar que el modelo que sostendrá la aplicación (en el caso de que exista trabajo con base de datos) es válido tanto para la versión a desarrollar como para posibles futuras evoluciones. Los cambios en el modelo son normalmente los que implican un mayor coste de adaptación en el desarrollo, al afectar a todos los aspectos de una aplicación. A partir de estas bases, existen funcionalidades que, por su sencillez, pueden no necesitar versiones evolutivas, u otras que por su complejidad en interacción pueden requerir prototipos que se pueden tomar como un paso intermedio al desarrollo evolutivo iterativo. Los prototipos no llegan a ser una versión del programa (porque no suelen ser nunca totalmente funcionales), pero pueden servir para sacar a la luz requerimientos no detectados que sobre un desarrollo final podrían ser muy costosos en tiempo y dinero.

Como buena práctica, se recomienda trabajar con el cliente en la fase inicial de diseño con *mockups*, **prototipos** y **pruebas simuladas de interacción**, de manera que se puedan adelantar problemas o necesidades a fases en las que el desarrollo ya está en marcha.

El prototipado y pruebas sobre él se pueden convertir en una mecánica muy útil y práctica en este tipo de aplicaciones, aprovechando lo mejor de los dos enfoques que hemos visto.

2. Planificación y estrategia

Antes de llevar a cabo cualquier proyecto (ya no solo dentro del mundo del desarrollo de software), es fundamental planificarlo y establecer una estrategia a seguir para lograr la consecución de los objetivos planteados. A continuación, analizaremos las diferentes acciones que hay que realizar para poder diseñar una estrategia a seguir a lo largo del ciclo completo del desarrollo de nuestras ARM, así como la manera de planificar ese desarrollo para poder establecer plazos y, por tanto, costes que sean lo más realistas posibles con el proyecto.

2.1. Análisis de requerimientos y funcional

Antes incluso de elaborar una estimación temporal del desarrollo (ya sea como valoración interna del esfuerzo que supondrá el proyecto, o como base para marcar un presupuesto para el cliente), es fundamental hacer un análisis funcional de la aplicación a desarrollar, para obtener los requerimientos necesarios a solventar y poder evaluarlos correctamente.

En muchas ocasiones, los proyectos no permiten dedicar el tiempo necesario a este apartado del ciclo de desarrollo de software, pero no por ello se ha de olvidar la vital importancia y repercusión que tendrá en todo el proceso de desarrollo posterior. Si queremos poder hacer una estimación temporal realista (y, por tanto, económicamente viable) de un proyecto, es vital haber detectado, antes de planificar el desarrollo, las funcionalidades principales y la clave de la aplicación.

El trabajo a realizar durante esta etapa consiste en comprender e interpretar las necesidades del usuario/cliente de nuestra aplicación y traducirlas en un documento claro y preciso en el que se establezcan las bases y pilares de las funcionalidades que ha de recrear nuestra aplicación. El proceso de comprensión e interpretación, en la mayoría de las ocasiones, implica la necesidad de aprender gran cantidad de información relativa al proceso de negocio del proyecto. En muchas ocasiones, el hecho de poder definir la mejor solución pasa por comprender a fondo el problema.

A menudo, también hay que concienciarse y concienciar al usuario/cliente de que, en el desarrollo del software, al igual que en cualquier otra área, los cambios a realizar, cuanto más tarde llegan, más problemas y esfuerzos implican.

Nunca se debería realizar una planificación sin basarse en funcionalidades precisas y reales de la aplicación a construir, ya que podría llevar a desfases importantes en el plazo de desarrollo. Cuanto más preciso sea el análisis, más desglosadas estarán las funcionalidades y, por tanto, menos margen de error existirá a la hora de calcular el tiempo de desarrollo asociado a cada una de ellas (y, generalmente, siempre que se habla de tiempo se habla de costes).

El análisis funcional tiene un claro peso en el inicio del desarrollo, pero no es algo que se hace una vez y queda estancado, sino que es importante su continua revisión para detectar lo antes posible cualquier funcionalidad que se haya pasado por alto en esa primera aproximación.

Para realizar esta etapa del desarrollo se suele establecer un calendario de reuniones con el cliente en el que se abordan las cuestiones críticas de la aplicación para, posteriormente, entrar en el detalle exhaustivo de cada una de ellas.

Es habitual que, antes incluso de realizar un presupuesto, se mantengan una o varias reuniones con el cliente (dependiendo de la complejidad del desarrollo) para determinar las funcionalidades y requerimientos base del proyecto, y con esa base hacer una estimación de tiempos y costes. Si la propuesta prospera, es cuando se lleva a cabo la planificación en detalle de reuniones para, por cada uno de esos requerimientos básicos, desglosar y desglosar al máximo las funcionalidades que se derivan de él.

Las reuniones se han de preparar con antelación y se ha de establecer, para cada una de ellas, un guion claro de los puntos a tratar y solventar, ya que el mayor enemigo de las mismas es la dispersión de objetivos a tratar (lo que puede convertirlas en inútiles si se generalizan los temas abordados). Una vez terminada, se ha de valorar si ha satisfecho todas las necesidades para su desarrollo y, en caso negativo, establecer el ajuste en el calendario correspondiente para finalizarla correctamente.

2.2. Definición de la estrategia (diseño de software)

Una vez que se cuenta con un conocimiento adecuado del proyecto a través del análisis funcional, se debe plantear la estrategia de desarrollo para solventar todas las funcionalidades requeridas de la mejor manera posible. Dentro de este concepto, englobaremos los siguientes puntos:

- Definición de la/s plataforma/s destino.
- Arquitectura / enfoque.
- Tecnología/s.
- Planificación.

2.2.1. Definición de la/s plataforma/s destino

Antes incluso de definir la arquitectura o enfoque de la aplicación, y una vez que se conocen los requerimientos y funcionalidades esperadas de la aplicación, es fundamental determinar la o las plataformas sobre las que se ha de ejecutar dicha aplicación. Esta definición de las plataformas objetivo o destino es una característica clave para la decisión posterior de la arquitectura, tecnología y planificación a desarrollar en el proyecto.

Actualmente, y pensando en ARM, el número de plataformas sobre las que se puede querer desarrollar una aplicación es extensa e incluso la decisión puede incluir varias de ellas de forma simultánea:

- Internet (en forma de RIA).
- Escritorio (Windows o Mac).
- Teléfonos (iPhone, Android, BlackBerry, Windows Phone...).
- Tablets (iPad, Android, BlackBerry...).
- Dispositivos (otros dispositivos como iPod, MP4s, Set Top Boxes...).
- Consolas de videojuegos.
- Televisión.

Debido a que las aplicaciones ARM están enfocadas al usuario, es habitual que se le quiera dar a este la posibilidad de emplearlas desde diferentes entornos o dispositivos. Por desgracia, no existe una tecnología que sea una panacea para resolver o abarcar todas estas posibilidades de forma unificada, de forma que es vital conocer las capacidades de cada tecnología en cada plataforma para poder tomar la decisión correcta sobre ese apartado una vez se han determinado las plataformas destino.

En el caso de la tecnología en la que se centra esta asignatura (Flash), el abanico de posibilidades es muy amplio y permite un trabajo bastante unificado en cuanto a desarrollo, e incluso diseño, dependiendo de la estructura y las necesidades gráficas de cada aplicación.

Como decíamos, hoy en día es muy habitual que todas las aplicaciones o servicios basados en datos quieran dar al usuario una solución en diferentes contextos y situaciones, de modo que no es nada extraño que existan diferentes clientes para las diferentes plataformas que aportan una misma base pero que se benefician de las características innatas del medio en el que corren.

Ved también

Véase, a este respecto, el apartado "Tecnologías relacionadas".

Ved también

Para conocer estas posibilidades, véase el apartado "Plataformas para publicación".

Existen muchos ejemplos de este tipo de aplicaciones; podríamos citar, como grandes referencias, Gmail, Facebook o Twitter. Estos servicios en la nube, llevados a las diferentes plataformas en forma de aplicaciones, se adaptan al medio en el que corren para mejorar la experiencia de usuario e integrarla con lo que un usuario espera en ellas. De esta forma, la interfaz del cliente de Gmail cambia radicalmente accediendo desde un terminal móvil de gama media, desde un *smartphone* como iPhone o desde un *tablet* como puede ser una BlackBerry Playbook. La aplicación de Gmail se convierte en una *webapp* en algunas de estas plataformas, o una RIA en otras. Incluso existen versiones clientes para escritorio desarrolladas por terceras partes.

Facebook y Twitter hacen algo similar, aunque en algunos casos van más allá e incluso crean aplicaciones nativas en algunas plataformas, o aprovechan tecnologías multiplataforma para cubrir otras de ellas.

En cualquier caso, cuando se desea dirigir una ARM a diferentes plataformas es cuando mayor fuerza cobra el hecho de emplear una **tecnología multiplataforma** como **Flash** o **HTML5** (para dar forma a una *webapp*), en vez de una tecnología nativa y específica.

El hecho de poder reutilizar grandes cantidades de código, unificar servicios en la parte de servidor en el caso de aplicaciones cliente-servidor, e incluso atacar a un rango de dispositivos diferentes dentro de una misma plataforma (como son todos los teléfonos móviles basados en Android, con sus diferentes pantallas, resoluciones, etc.), se convierten en una ventaja definitiva cuando no se tienen recursos ilimitados para desarrollar.

En el caso de **HTML5**, es obvio que se emplea el navegador para unificar el comportamiento de las diferentes aplicaciones en diferentes terminales. Digamos que el navegador se convierte en una especie de “máquina virtual” que permite unificar, en gran medida, el trabajo realizado en un lenguaje común. Esto tiene limitaciones claras que, en algunos casos, se pueden solventar mediante aplicaciones terceras (como PhoneGap) que extienden las capacidades del lenguaje fuera del navegador, mediante “puentes” que se comunican con funcionalidades nativas en cada aplicación.

En el caso de **Flash**, Adobe AIR es el que hace la magia para que, a partir de un mismo código, se pueda ejecutar en diferentes plataformas, incluso en el caso del sistema operativo iOS, y sea “re-compilado” a código nativo para comportarse como tal.

A pesar de que el rendimiento, en algunos tipos de aplicación, es inferior cuando se emplea una tecnología multiplataforma (por ejemplo, juegos con alta carga gráfica o 3D), en la mayoría de casos, el beneficio de un desarrollo común para varios dispositivos es superior a la posible pérdida de rendimiento que, en muchos casos y tipologías de aplicaciones, es directamente inexistente.

Ved también

Hemos hablado de PhoneGap en el apartado “Tecnologías relacionadas”.

Es resumen, podemos afirmar que en muy pocos casos un desarrollo que tiene como objetivo más de una plataforma es asumible con tecnologías nativas, y es por esta razón por lo que las aplicaciones ARM (que casi siempre están ligadas a tipos de contenidos y servicios que pueden tener sentido en varias plataformas) se suelen desarrollar mediante tecnologías multiplataforma como Flash o HTML5, ya que este tipo de aplicaciones suelen basarse en servicios en la nube y, por tanto, los perfiles de desarrollo incluso se pueden reaprovechar entre las diferentes vistas o clientes de la aplicación.

2.2.2. Arquitectura/enfoque

En el desarrollo de cualquier aplicación es fundamental analizar cuál puede ser la mejor arquitectura que la soporte y qué enfoque se ha de aplicar para obtener la mejor solución. La mejor solución no es necesariamente la que se refiere a aspectos relativos a la calidad de la programación (que son importantes), sino aquella que responda de forma más eficaz y eficiente a las necesidades impuestas por el proyecto.

Con este objetivo en mente, se ha de definir qué tipo de arquitectura requiere nuestra aplicación. Dependiendo de su complejidad, puede requerir diversos mecanismos y componentes tales como: desarrollo del lado de servidor, servicios web o API de comunicación, mecanismos de almacenamiento persistente, integración con hardware o sistema operativo, etc.

En el caso de que la aplicación requiera de un desarrollo del lado de servidor, cosa muy habitual en la actualidad, se deben estudiar las necesidades de dicho componente de la arquitectura, que suele ser crítico en el funcionamiento del servicio. Aspectos como capacidad de proceso (por ejemplo, para tareas de conversión de imágenes o vídeo), capacidad de conexiones concurrentes, bases de datos, ancho de banda, reparto de carga, escalabilidad, etc., son de obligatorio estudio en este tipo de soluciones.

Es fundamental determinar todos estos puntos clave de la aplicación, ya que en muchos casos pueden ser bloqueantes frente a determinadas tecnologías, o puede darse también que alguna tecnología no sea capaz de ofrecer la capacidad/rendimiento exigida a la aplicación.

2.2.3. Tecnología/s

La elección de la tecnología o tecnologías no ha de ser una decisión arbitraria del desarrollador, sino que debe venir dada por las funcionalidades del proyecto que implican una arquitectura concreta. Un buen analista no debe adaptar el proyecto a la tecnología, sino elegir la mejor tecnología en función de las necesidades del proyecto.

Para poder tomar, por tanto, estas decisiones con garantías de éxito, es importante conocer las diversas tecnologías existentes, así como sus puntos fuertes y débiles.

En el caso de ARM, este aspecto cobra aún mayor importancia, ya que no todas las tecnologías soportan todos los posibles formatos de un proyecto, y lo que es más importante, no todas son capaces de trabajar con dichos formatos con la misma agilidad, rendimiento y complejidad.

En muchas ocasiones, se entiende que las decisiones tecnológicas tienen que estar centradas principalmente en el rendimiento, pero este enfoque, que es muy cierto en la teoría, no lo es tanto en la práctica. Los proyectos reales tienen muchos factores que influyen en los mismos, entre los que destacan:

- Funcionalidades y prioridades sobre las mismas.
- Tiempo de desarrollo.
- Costes de desarrollo.
- Evolución y mantenimiento.
- Versiones en diferentes plataformas.

Dando por hecho que varias tecnologías son capaces de solventar las funcionalidades de una aplicación con un rendimiento aceptable, entran en juego el resto de factores que en la realidad se convierten en tan críticos como el primero.

Elegir una tecnología ha de permitir agilizar los tiempos de desarrollo (que generalmente implican mejoras en los costes de producción) y simplificar los procesos futuros de evolución y mantenimiento.

Además, hoy en día es muy habitual que se quiera disponer de al menos dos versiones de la aplicación, de modo que las capacidades de reutilización de conocimiento y código entre versiones se convierten en otro factor clave para el tiempo y los costes.

Ved también

Hemos visto estos aspectos en el apartado "Tecnologías relacionadas".

Ved también

Véase el apartado "Definición de la/s plataforma/s destino".

Veamos un ejemplo: se desea realizar una aplicación con una arquitectura cliente-servidor, con toda la información en la nube y una API que permita gestionarla para ser mostrada en diferentes plataformas.

Hoy en día, casi cualquier tecnología podría permitir realizar esa comunicación con el servidor y el renderizado de la información de forma adecuada. Una vez solventadas las funcionalidades críticas es cuando entran en juego factores como cuál es más ágil en función de cómo se muestran los datos, cuál permite desarrollos más rápidos, cuál tiene una base de desarrolladores mayor (para tener garantías de evolución y mantenimiento, al no requerir de perfiles complicados de contratar), cuál permite trabajar con las mismas herramientas y lenguajes aunque se deseen sacar versiones para diferentes plataformas, etc.

Como se ha querido ilustrar en el ejemplo, una vez aseguradas (con la solvencia deseada) las funcionalidades críticas de una aplicación, entran en juego muchos factores que nada tienen que ver con la teoría pura de la programación, y mucho con la **productividad, simplicidad y rendimiento** de los recursos invertidos.

2.2.4. Planificación

Cuando en el proceso de desarrollo se ha definido una arquitectura y un enfoque para obtener la solución, y además se ha establecido una tecnología para llevar a cabo el proyecto, es momento de planificar cómo va a ser dicho desarrollo.

Dentro de la planificación, no solamente hay que valorar el tiempo de desarrollo que conllevarán las tareas a realizar con los recursos disponibles, sino que es importante tener en cuenta las relaciones que pueden existir entre las diferentes etapas y funcionalidades, ya que en muchas ocasiones, para poder abordar unas, es necesario tener completadas otras previamente.

Una buena manera de realizar la planificación es agrupar las tareas por los diferentes perfiles del equipo que trabajará en el proyecto y repartirlas de manera que se minimicen las dependencias temporales. Por ejemplo, mientras que se lleva a cabo el diseño de interfaz, se podría perfectamente trabajar en el modelo de datos (base de datos) y en los servicios web que permitan trabajar con esa información. De esta forma, cuando el diseño esté aprobado, una parte de la carga de desarrollo estará ya solventada y será más fácil contar con recursos para dedicar a la implementación de la parte cliente (que suele ser la que tiene el mayor peso visual).

Es una práctica muy recomendable desglosar, de forma meticulosa, las funcionalidades a desarrollar para que los tiempos asignados a cada una de esas tareas sean lo más cortos posible y exista, por tanto, una mayor flexibilidad de organización para encajarlas en el tiempo. Además, el hecho de disponer de tareas concretas de duración más corta hace que el margen de error en la valoración temporal de las mismas se reduzca, resultando normalmente en planificaciones más exactas y con más capacidad de control y seguimiento de la evolución de dichas tareas.

3. Diseño, usabilidad, experiencia de usuario y accesibilidad

A partir de la definición de funcionalidades realizada en base al análisis de requerimientos, se han de extraer las necesidades visuales y de interacción de la aplicación, un aspecto que en muchas ocasiones marcará y podrá condicionar la forma de afrontar la codificación de algunos módulos de la aplicación, sobre todos aquellos relacionados con la interacción con el usuario.

En el desarrollo de ARM, el trabajo sobre el diseño, usabilidad y experiencia de usuario tiene un peso muy importante en el resultado final del producto. Las ARM están totalmente enfocadas al usuario y, por tanto, estas tres cuestiones son vitales si se quiere lograr el éxito.

Para el trabajo sobre ellas, existen una serie de técnicas y metodologías que permiten agilizar el proceso, abstraer el cliente en determinadas fases para que se centre en lo verdaderamente importante y lograr, de esta manera, sacar el máximo partido de su conocimiento.

El prototipado, los patrones de diseño y el conocimiento del usuario y de sus necesidades son las herramientas y mecanismos de las que disponemos para preparar el proyecto hacia un diseño adecuado.

3.1. Prototipado. Definición y concepto

Siguiendo con la planificación del desarrollo de la aplicación, se comienza el desarrollo de la misma. Este desarrollo no hace solamente alusión a la parte dedicada a la programación, sino en general a la producción de la solución a crear.

Dentro de la producción, se engloba el **diseño** y la **programación**, y en ambos campos es una práctica habitual y deseable la elaboración de un **prototipo**.

Los **prototipos** son una representación limitada de un referente final que permite explorar el uso de dicho referente antes de invertir el esfuerzo necesario en convertirlo en una realidad, permitiendo así un proceso iterativo que genera calidad.

Advertencia

Este apartado no hace alusión al diseño del sistema, que ya se ha tratado en puntos anteriores, sino a la fase o proceso de diseño de la parte visual de la aplicación, que aunque no es un componente clásico del ciclo de desarrollo (entendiendo como desarrollo la programación del sistema), en el caso de las ARM tiene suficiente relevancia y peso como para tratarlo como parte de él.

Los diferentes puntos que se tratan en este apartado han de ser enfocados de manera individualizada para cada plataforma (en el caso de una aplicación multiplataforma), teniendo en cuenta patrones y convenciones que existen en cada una de ellas y a las que el usuario está acostumbrado. En las referencias finales se incluyen una serie de guías para las plataformas móviles más extendidas.

Un prototipo puede ser cualquier cosa, desde un trozo de papel con sencillos dibujos a una versión simplificada pero parcialmente funcional de la aplicación. Lo importante es disponer de una representación sencilla y ágil frente a cambios sobre la que simular el futuro comportamiento de nuestra aplicación.

Un ejemplo claro de la importancia del prototipado se da en el diseño industrial, en donde los prototipos pueden suponer el éxito o fracaso de la producción de un producto.

En el caso del **diseño**, la creación de prototipos aporta una serie de cualidades que son fundamentales para optimizar el posterior desarrollo, e incluso para facilitar el detalle final gráfico:

- Permite comunicar de forma mucho más clara las ideas que existen tras el diseño.
- Aísla los aspectos que interesan del diseño de las distracciones que acarrea una visión conjunta y completa.
- Ayuda a definir la estructura de los contenidos.
- Establece las bases de la apariencia final, lo que permite al diseñador centrarse en los detalles gráficos una vez aprobado.
- Agiliza los cambios en los ciclos iterativos al trabajar sobre modelos gráficos simples.
- Permite analizar y decidir sobre el papel la interacción entre elementos de una aplicación, y hacerlo dentro de su contexto.

Cuando estamos enfocando nuestro proyecto a la creación de una ARM, el prototipado tiene aún mayor relevancia, ya que estas aplicaciones, por norma general, tienen un alto grado de interacción con el usuario (debido a la necesidad de mostrar diferentes tipos de contenidos). Los prototipos podrían quedarse solamente en la forma de mostrar los contenidos, pero sería desaprovechar tremendamente sus posibilidades a la hora de poder analizar las interacciones y los cambios de pantallas tal y como los haría el usuario sobre la aplicación real.

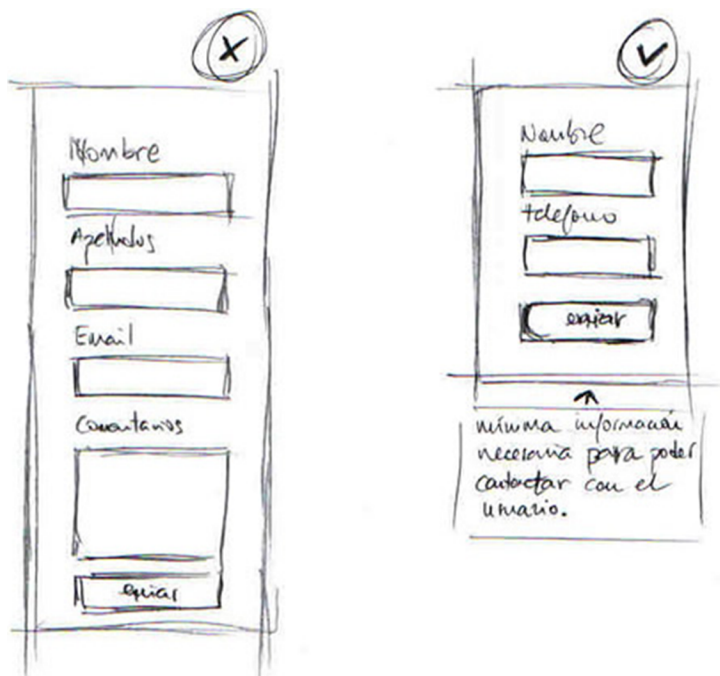
Es habitual trabajar objetivos concretos que el usuario puede plantearse sobre los prototipos para detectar situaciones no deseables o la necesidad de añadir elementos nuevos que agilicen las tareas críticas y habituales. El hecho de trabajar estas mecánicas sobre prototipos permite minimizar los costes de los cambios y reducir la detección de estos mismos problemas en fases posteriores, lo que siempre implica mayores impactos en el desarrollo.

3.2. Tipos de prototipos

Dentro del proceso iterativo de prototipado pueden emplearse diferentes tipos de prototipos, cada uno con unas cualidades específicas que lo hacen más apto para las diferentes etapas de esta fase.

1) Prototipado en papel

La creación de prototipos en papel suele tener lugar en las fases más tempranas del diseño, sobre todo, en la conceptualización del servicio. Puede servir para trabajar sobre la estructura visual de los diferentes elementos de la aplicación, probar la funcionalidad (simulando la navegación entre diferentes pantallas) y la interacción (planificando cómo se va a producir a partir de las acciones del usuario).



Wireframe en papel a mano alzada

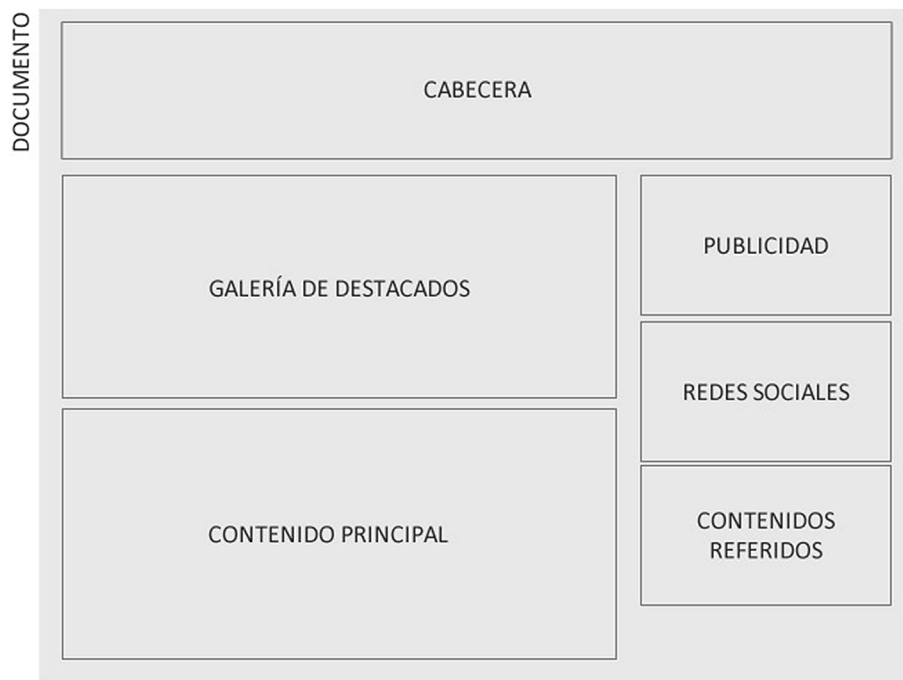
Este tipo de prototipos no persigue el detalle en el diseño o la fidelidad al producto final, sino que aísla al máximo las bases de la aplicación de esos aspectos para que en el trabajo sobre el prototipo no interfieran. La información que muestran son ejemplos que en principio nunca persiguen ser reflejo de la información final, salvo en la consideración de los espacios necesarios para representar los contenidos que albergarán en su definición final.

Por ejemplo, la zona reservada para el texto del detalle de una noticia se elaborará teniendo en cuenta la extensión habitual de este elemento.

Los cambios y la agilidad al cambio son máximas, lo que lo convierte en el mejor formato para las fases de conceptualización y trabajo en grupo.

2) Wireframes de bajo nivel

Este tipo de prototipos son empleados, principalmente, para definir la estructura de las pantallas y la distribución espacial de los contenidos en ellas, las zonas de navegación, las zonas principales de contenidos, etc. Como habitualmente suelen realizarse con software informático, no son prácticos para trabajar interacciones o funcionalidad, ya que no permiten realizar cambios de forma ágil ni trabajar con ellos aplicando variaciones sobre la marcha.



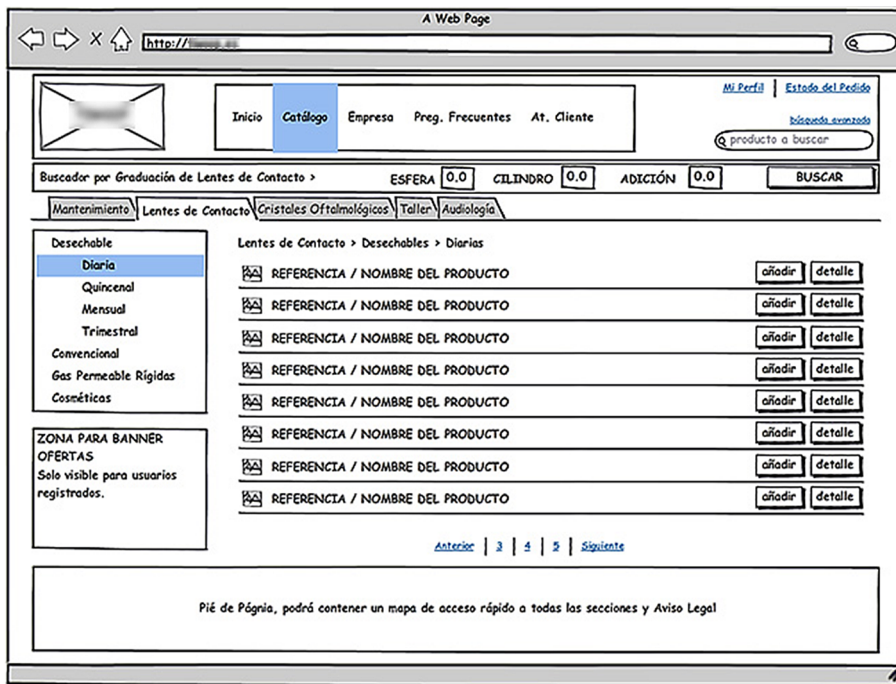
Wireframe de bajo nivel

Este tipo de prototipos se suele reducir a cajas que delimitan zonas para contenidos en cuyo interior hay etiquetas que indican qué contenido albergarán.

3) Wireframes detallados

Los *wireframes* detallados son la evolución de los *wireframes* de bajo nivel e incorporan mucho más detalle sobre las zonas previamente establecidas.

En estos prototipos ya se entra en la definición de los elementos interiores de cada zona, mecanismos de interacción y determinada funcionalidad, aunque siguen adoleciendo de poca capacidad de cambio, por lo que es más complicado usarlos para definir los comportamientos de las pantallas frente a las acciones de los usuarios.



Wireframe o mockup detallado

En la elaboración del detalle de cada zona se tienen en cuenta posibles contenidos cercanos a la realidad para anticiparse a problemáticas relacionadas con la extensión de los mismos, y de esta manera contemplar sistemas de paginación, *scroll*, etc.

Suelen elaborarse para una misma pantalla diferentes versiones en función de algunas interacciones (botones, barras de navegación, pantallas emergentes, cuadros de diálogo, etc.) para poder analizar la evolución de la misma a través de ellas.

4) Prototipo visual

Este prototipo es el que tradicionalmente se ha entendido como diseño previo. Sobre los *wireframes* evolucionados y confirmados, se aplica una iteración adicional de detalle que permite trabajar ya la parte puramente estética de los elementos definidos en las anteriores iteraciones.



Diseño detallado para propuesta aplicación tablet

Gracias a haber fijado la estructura en los *wireframes* de bajo nivel, y los elementos e interacciones en el prototipo en papel y *wireframes* de alto nivel, el diseñador puede centrar sus esfuerzos en el detalle gráfico de los elementos para complementar y reforzar la funcionalidad de estos.

En este prototipo, se potencia todavía más la fidelidad de los contenidos mostrados, para tratar de aproximar al máximo la apariencia al resultado final de la aplicación.

3.3. Patrones de diseño

Al igual que existen los denominados patrones de diseño (http://en.wikipedia.org/wiki/Software_design_pattern) en programación, que aportan soluciones de calidad a problemas repetitivos y comunes, existen patrones de diseño que resuelven situaciones comunes que se dan a la hora de mostrar la información y permitir interactuar con ella.

El diseño de interfaces está muy ligado al usuario y a los comportamientos y hábitos que este tiene frente a situaciones que se suelen repetir a la hora de trabajar con programas. Los patrones de diseño se ubican justo en el centro de la combinación de diseño e interacción, potenciando la usabilidad de nuestras aplicaciones.

Los patrones de diseño y las convenciones heredadas pueden facilitar enormemente la curva de aprendizaje de los usuarios a la hora de manejar nuestras aplicaciones. Además, pueden ahorrarnos mucho tiempo a la hora de idear soluciones para representar modelos de información en pantalla, con alternativas que están ampliamente probadas. No obstante, patrones y convenciones no son lo mismo.

Una **convención** heredada es, por ejemplo, que el logotipo de una aplicación web te dirija al inicio de la misma, o que el botón de cerrar en una aplicación Windows esté en la parte superior derecha de una ventana.

Un **patrón de diseño** aporta una solución a una problemática existente, relacionada no solo con acciones directas y simples, sino con la organización de la información e interacciones complejas.

Un ejemplo sería la conocida “miga de pan” en las navegaciones jerárquicas en profundidad, que nos permite tener en un espacio reducido el camino completo a lo largo de la jerarquía, y además representar la dirección de ese anidamiento. Este patrón lo podemos ver tanto en un catálogo web como en los exploradores de archivos de sistemas tan extendidos como Windows y Mac OS.

3.4. Ejemplos de patrones de diseño

A continuación, vamos a ver unos ejemplos de patrones de diseño de uso extendido que aportan soluciones inteligentes a problemas complejos. Al final de estos ejemplos, se ofrece un apartado con referencias a diferentes recursos donde poder encontrar información más exhaustiva sobre esta temática.

3.4.1. Interacción básica

Dentro de la interacción básica, vamos a ver dos elementos que son especialmente relevantes por la cantidad de veces que se dan en los desarrollos: los **botones de acción** y los **sistemas de paginación**.

Botones de acción

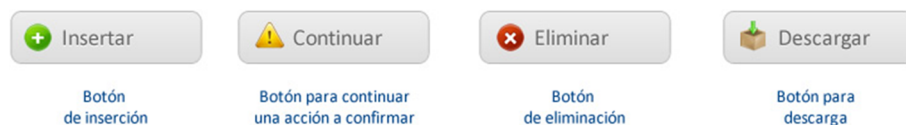
Cuando se quiere resaltar una acción directa al usuario (añadir, eliminar, cancelar, insertar...), normalmente se usa el concepto de *botón* porque, en el mundo digital, **los botones representan siempre una acción**. Dentro del campo de los botones existen muchas alternativas para reforzar la acción a realizar e incluso para dirigir la atención al usuario frente a dos opciones.

Los botones clásicos, que como se puede apreciar en la imagen presentan algunas diferencias dependiendo del navegador y del sistema operativo, simplemente por su forma transmiten al usuario directamente el concepto de realizar una acción. Normalmente, el botón se etiqueta incluyendo una forma verbal para reforzar ese hecho.



Ejemplos de botones convencionales

Para reforzar la acción de la etiqueta del botón, en muchas ocasiones, se le acompaña un icono que transmita el mismo concepto, de manera que visualmente se está transmitiendo por dos vías un mismo concepto.



Ejemplos de botones con icono de apoyo

En ocasiones, incluso se omite la etiqueta cuando por cuestiones de espacio no es posible mantener la estructura, dejando solamente el icono como representación del botón. Un lugar muy habitual donde encontramos este patrón es en zonas donde se ofrecen varias acciones sobre elementos de un listado.

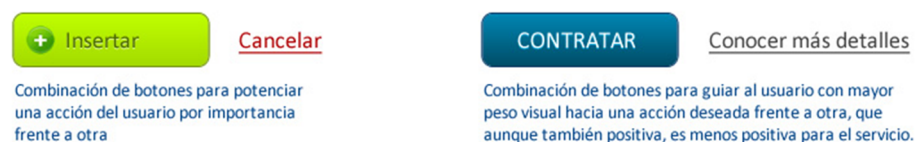


Otra forma de resaltar el concepto detrás de un botón es emplear colores que apoyen el tipo de acción que se ofrece. Se suelen usar cuando se está hablando de acciones positivas o negativas (por ejemplo: aceptar / cancelar).



Ejemplos de botones con refuerzo por color

Finalmente, vemos una combinación de botones para potenciar aún más si cabe la acción prioritaria frente a dos enfrentadas. Un caso muy común es el que se muestra en la imagen, donde se reduce la carga visual de la acción más restrictiva, incluso combinando botón y enlace, en vez de dos botones.



Ejemplos de combinación de botones

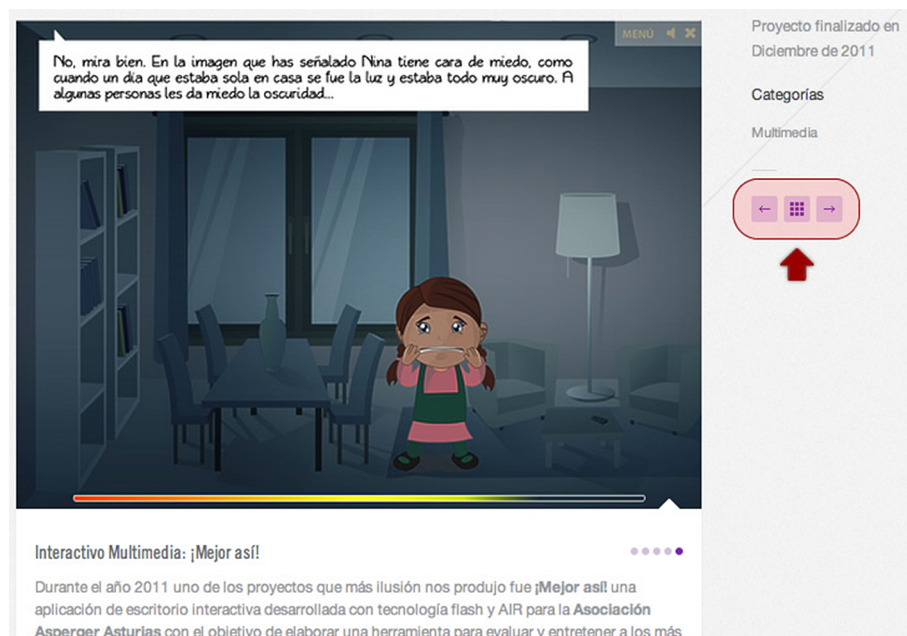
Mecanismos de paginación

Un elemento muy habitual, tanto en aplicaciones web como en aplicaciones de escritorio o en movilidad, es la paginación. La paginación no está reñida con los sistemas de *scroll*, sino que se pueden complementar e incluso integrar, como ocurre actualmente en listados en dispositivos móviles como los basados en iOS o Android, donde el último elemento de la lista se convierte en el acceso a la siguiente página de resultados (habitualmente carga otro bloque de resultados que se añaden a la lista sin perder el punto de *scroll* en el que nos encontramos).

Vamos a ver algunos ejemplos de navegación existentes y que tienen más o menos sentido en función de la cantidad de resultados o expectativas del usuario tras ellos.

1) Paginación convencional siguiente y anterior

Esta paginación es habitual en sistemas donde se está mostrando un elemento en detalle y no un listado (en la imagen, se puede ver el sistema enmarcado en la parte derecha, con un botón anterior, un botón siguiente y un botón central para acceso al listado en formato de rejilla). Es una paginación secuencial que permite navegar entre elementos para poder ir viendo el detalle de cada uno de ellos.

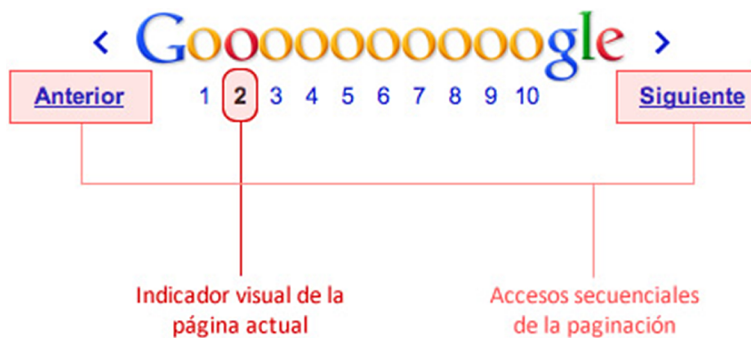


Navegación siguiente y anterior, con acceso general

Es interesante destacar que, por norma general, en las aplicaciones occidentales el botón anterior va a la izquierda y el botón siguiente a la derecha, dando la sensación de que el contenido está en medio. Esto no es una casualidad, sino que encaja con el patrón que tiene el usuario de lectura de izquierda a derecha, donde la izquierda es anterior a la derecha.

2) Paginación convencional numérica

Esta paginación es mucho más adecuada para listados de resultados, no para detalles, y persigue el concepto de “una página entre varias”. Se muestra un conjunto de resultados en pantalla (página actual) y se permite navegar a otras páginas previas o posteriores de forma directa. Normalmente, ofrece varias páginas próximas y el cierre se suele representar con la primera y la última, además de indicar, mediante unos puntos suspensivos, que existen más páginas de las que se pueden visualizar.



Ejemplo de paginación numérica

Otra opción es la paginación por rangos, donde en vez de representar “páginas”, cada elemento indica el orden de resultados que contiene en su interior.



Ejemplo de paginación por segmentos

3) Paginación alfabética

Este tipo de paginación se emplea cuando los contenidos son sensibles ante este criterio. Cada letra representa una página del conjunto y en cada página se muestran todos los resultados que están incluidos en la clasificación alfabética. Este patrón se emplea de forma habitual en la aplicación de contactos de los teléfonos móviles, por ejemplo, o en glosarios como el que se puede intuir en el ejemplo de la imagen.



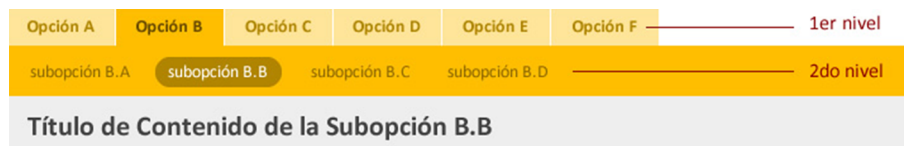
Ejemplo de paginación alfabética

4) Navegación

Esta área del diseño de aplicaciones es una de las más complicadas de gestionar en muchos casos, de modo que es una de las áreas donde mayores esfuerzos se han hecho para facilitar al usuario el acceso a los contenidos. Entre los diferentes patrones de diseño que existen en este apartado, vamos a analizar la navegación por pestañas y el *dashboard/springboard* en dispositivos móviles.

a) Navegación por pestañas anidadas (*double tab navigation*)

Como podemos observar en la siguiente imagen, la navegación por pestañas acompañada de un diseño coherente permite acceder, de una forma ágil, a varios contenidos pertenecientes a un mismo bloque de información o entidad. En el caso de que la primera clasificación de contenidos no sea suficiente y dentro de cada uno de ellos sea necesaria una segunda navegación, este sistema permite añadir una segunda línea de menú para estas subcategorías.



Navegación por pestañas anidadas

Se puede observar que, gracias a los colores y al acabado gráfico, se transmite claramente que los contenidos que se visualizan pertenecen al apartado de la pestaña y la subopción seleccionada. Además, la navegación por pestañas envuelve el contenido potenciando aún más la sensación de conjunto.

b) *Dashboard/springboard* en dispositivos móviles

Este patrón es ampliamente conocido por ser el menú principal de acceso a aplicaciones de iOS (iPhone e iPad) y, con ligeras variaciones, en Android. Es un patrón que ya existía en algunos apartados de algunos teléfonos Symbian y, básicamente, muestra un menú en rejilla basado en iconos y texto.

Este patrón ha sido llevado a las aplicaciones y empleado para dar acceso directo a sus apartados siguiendo la misma filosofía. En el caso de aplicaciones para dispositivos (y máxime si los dispositivos pueden tener diferentes tamaños de pantalla), es un patrón realmente útil y positivo para la usabilidad de la aplicación, razón por la que ha sido ampliamente adoptado.



Patrón dashboard

Como se puede ver en la imagen, el trabajo en rejilla mediante el uso de iconos permite tener un comportamiento altamente flexible ante las variaciones de tamaño de pantallas (resolución) y ante diferentes volúmenes de información.

La rejilla se puede combinar bien con paginación (como es el caso de iOS, con la navegación lateral), bien con *scroll* (como es el caso del menú principal de aplicaciones de Android), o incluso combinando los dos sistemas, como ocurre en el escritorio del dispositivo PlayBook de BlackBerry.

Esta magnífica capacidad de adaptación frente a diferentes pantallas y cantidad de opciones lo han convertido, actualmente, en el rey de los formatos de navegación en dispositivos táctiles.

5) Procesos

Tanto en aplicaciones para la web como en aplicaciones de escritorio o dispositivos, es frecuente encontrarse con situaciones en las que se requiere realizar un proceso, ya sea para configurar una búsqueda, enviar información, etc. Se ha constatado que, cuando los procesos son demasiado extensos, el usuario tiene una tendencia a frustrarse y a abandonar durante el proceso, o incluso a no iniciarlos al ver la perspectiva global.

Ante esta situación, y sobre todo en dispositivos móviles, es necesario minimizar la cantidad de esfuerzo que ha de hacer el usuario para completar estos procesos y reducir al máximo los posibles errores que pueda cometer al introducir información. Además, se le ha de informar del punto en el que se encuentra en todo momento, lo que ha dado lugar a un patrón de diseño deno-

minado *Wizard* o asistente, que se base en dividir la información a solicitar en pasos cortos y sencillos, y en acompañar estos pasos de una zona indicadora de progreso que permita al usuario ver que efectivamente avanza.

Patrón *Wizard* multipágina

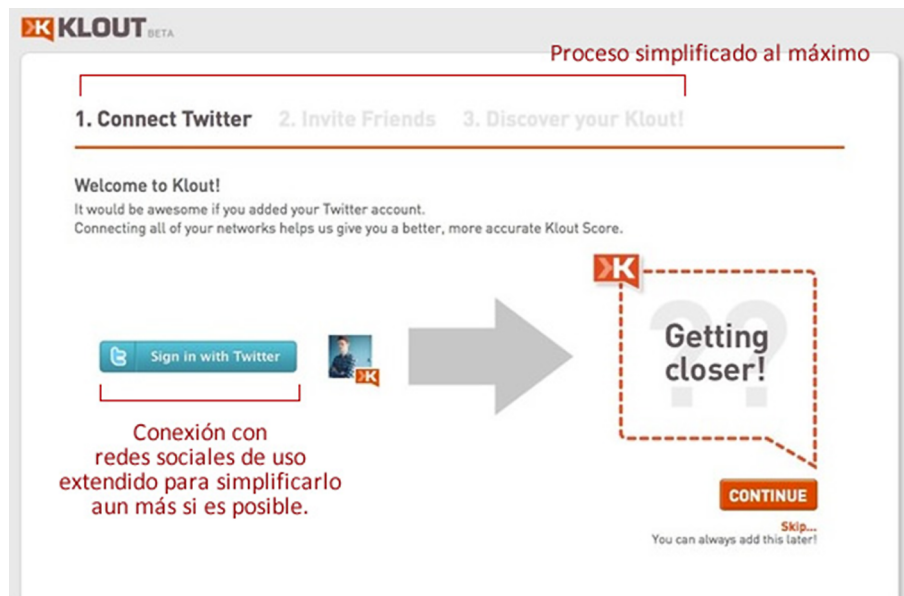
El número de pasos no puede ser elevado, ya que la barra de progreso frustraría más que animaría, y el contenido de los mismos tampoco puede ser demasiado denso. Es interesante que la barra de progreso indique claramente lo que se encontrará en los siguientes pasos y la dirección que sigue el usuario en dicho proceso (bien con indicadores, bien con una secuencia ordenada).

Patrón *Wizard* en una sola página

En muchas ocasiones hay que filtrar los datos a solicitar y quedarse con los que son exclusivamente necesarios, sobre todo cuando son procesos en los que el usuario no tiene un interés personal en completarlos.

Por ejemplo, un formulario de registros o de completado de un perfil, frente a un formulario de un configurador de un coche, en el que el resultado es justo lo que el usuario quiere obtener.

Hoy en día, en el campo de la solicitud de datos, la tendencia clara es a reducir al máximo este proceso al usuario, ofreciéndole incluso sistemas de identificación ya existentes basados en conexión con Facebook, Gmail, Twitter, etc. De esta manera, se minimiza el esfuerzo del usuario para darse de alta o participar, y posteriormente se le ofrece completar su perfil o configurar de forma avanzada su cuenta.



Patrón Wizard simplificado con API de redes sociales

3.4.2. Otras referencias

Estas son algunas referencias existentes en la Web relativas a los patrones de diseño a la hora de trabajar formas de mostrar los contenidos, interactuar con el usuario, etc. Es importante tener en cuenta que los patrones de diseño no solo hacen alusión a la presentación estática de los contenidos, sino que en la mayoría de los casos se adentran en los aspectos psicológicos de la interacción del usuario con dichos contenidos, favoreciendo siempre la sencillez de uso, el conocimiento anticipado de los resultados y mostrando el camino a seguir para encontrar la información demandada (o en algunos casos, dirigiendo al usuario hacia la que se le desea mostrar).

Patrones de diseño generales:

<http://www.welie.com/patterns/>.

<http://billwscott.com/share/presentations/2007/rwe/Antipatterns.pdf>.

<http://developer.yahoo.com/ypatterns/>.

<http://designingwebinterfaces.com/designing-web-interfaces-12-screen-patterns>.

<http://www.smashingmagazine.com/2009/06/15/40-helpful-resources-on-user-interface-design-patterns/>.

<http://designinginterfaces.com/patterns/>.

<http://patternry.com/patterns/>.

<http://www.patternbrowser.org/code/pattern/pattern.php?4,1,0,1,0,6.php>.

<http://www.webdesignpractices.com/>.

<http://www.userplus.com/blog/design-patterns/>.

<http://ui-patterns.com>.

Showcases de patrones aplicados:

<http://patterntap.com>.

<http://konigi.com/interface/latest>.

http://www.smileycat.com/design_elements/.

Dispositivos móviles:

<http://www.androidpatterns.com/>.

<http://mobile-patterns.com>.

<http://mobiledesignpatterngallery.com>.

Guías de estilo y patrones para diferentes plataformas móviles:

<http://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html>.

http://developer.android.com/guide/practices/ui_guidelines/index.html.

<http://msdn.microsoft.com/en-us/library/hh202915%28v=VS.92%29.aspx>.

3.5. Usabilidad

El constante aumento de creación de Aplicaciones Rich Media ha elevado a otro nivel el estudio y conocimiento de dos disciplinas íntimamente relacionadas aunque no iguales: **usabilidad** y **experiencia de usuario**.

La usabilidad se refiere a la facilidad con la que un usuario logra sus objetivos a través de una herramienta.

La usabilidad es un concepto que coexiste con cualquier herramienta que tenga una función, lo que por supuesto incluye todas las que se encuentran fuera del concepto de *digital*. Conceptos como *diseño*, *ergonomía*, etc. están íntimamente ligados con la usabilidad, ya que pueden potenciar de manera notable la facilidad de uso de una herramienta, ya sea mediante su forma, mediante la comunicación o diálogo no verbal que establece con el usuario (diseño) o mediante cualquier otra característica.

En el caso de las aplicaciones o software, la usabilidad cobra especial importancia, pues el usuario trabaja sobre elementos no tangibles. En este campo, la usabilidad se tiene que centrar en la **comunicación** con el usuario, en la **interacción** con este, tanto para guiarle en su proceso de uso como ayudarle en todo momento a saber en qué punto se encuentra y hacia dónde va. El estudio de la usabilidad está muy ligado a la psicología y a los estudios estadísticos basados en la **experiencia**. Actualmente, existe una rama del desarrollo de software centrada en el estudio y análisis de esta característica, para mejorar las áreas en las que se detectan lagunas por medio de estudios con usuarios reales.

Hoy existen incluso técnicas de *eye tracking* para determinar no solamente las zonas en las que el usuario interactúa, por ejemplo con el ratón, sino incluso para definir las zonas en las que el usuario inicialmente busca encontrar su objetivo.

En general, se dispone como regla básica para mejorar la usabilidad la creación de interfaces de usuario simples, intuitivas y con una curva de aprendizaje muy reducida. En muchas ocasiones, las convenciones establecidas y detectadas en los estudios son un camino muy válido para ayudar al usuario a que el aprendizaje de uso sea menor, ya que aplicar criterios ampliamente extendidos y generalizados ayuda a tener recursos incluso sin haber usado antes la aplicación.

Ved también

La usabilidad se trata con mayor profundidad en la asignatura *Diseño de interfaces multimedia*. Puede ser interesante repasar los materiales de la asignatura como complemento a este material.

Enlaces relacionados

<http://es.wikipedia.org/wiki/Usabilidad>.
<http://www.usability.gov/>.
http://albertolacalle.com/hci_articulos.htm.
<http://happyuser.xperienceconsulting.com/>.
<http://useit.com>.

3.6. Experiencia de usuario

Lectura recomendada

Steve Krug (2006). *No me hagas pensar. Una aproximación a la usabilidad en la web*. Madrid: Prentice-Hall.

La experiencia de usuario está relacionada con la usabilidad, aunque va mucho más allá del mero hecho de ayudar al usuario a lograr sus objetivos, ya que se centra en transmitir al usuario determinadas sensaciones para que su percepción de las interacciones con la aplicación sea agradable y positiva.

Con este objetivo, es lógico deducir que la experiencia de usuario se aplica tanto en el propio diseño como en la interacción, persiguiendo aportar al usuario sensaciones positivas de uso.

Para transmitir las sensaciones deseadas, es importante conocer la psicología del usuario y trabajar en todos los factores necesarios que puedan desencadenar esa sensación positiva.

Una plataforma donde se ha llevado al extremo esta característica son los sistemas operativos de los dispositivos móviles. Una clave de la experiencia de usuario, para determinar su satisfacción, es la “fluidez” de la interfaz y de las interacciones. Pantallas con una gran sensibilidad y una gran nitidez combinadas con animaciones suaves y equilibradas ante las interacciones logran transmitir un “mejor funcionamiento”, que puede ser más visual que real, pero que puede incluso disimular ante el usuario tiempos de carga, renderizado de imágenes, etc.

3.7. Equilibrio entre usabilidad y experiencia de usuario

Es evidente que usabilidad y experiencia de usuario deben coexistir para lograr que la aplicación cumpla las expectativas del usuario y transmita, además, una sensación agradable en el uso. La ausencia acusada de alguno de estos dos aspectos puede llevar a la frustración, bien por la incapacidad de lograr los objetivos deseados, bien por existir una mala sensación tras el uso de, por ejemplo, la forma de visualizar la información o de interactuar con ella... A fin de cuentas, a nadie le gustan las cosas feas o toscas hoy en día.

El desarrollo web ha evolucionado mucho en los últimos años en la dirección correcta, mejorando los procesos que habitualmente son tediosos, complicados y en muchos casos fallidos. El mayor soporte para estándares por parte de los navegadores, la evolución de los lenguajes y el aumento del conocimiento del medio por parte de los equipos de desarrollo, han permitido que procesos como la subida de archivos, ordenaciones de elementos, etc., puedan realizarse en muchos sitios web con acciones tan simples y lógicas para el usuario como “arrastrar y soltar”.

Usabilidad no tiene por qué ser sinónimo de *falta de diseño*, y experiencia de usuario no debe ser sinónimo de *todo vale*.

En la creación de aplicaciones es recomendable trabajar en la línea de la facilidad inicial de para que el usuario realice las funcionalidades críticas (usabilidad), y posteriormente ampliar las mismas con mejoras en la interacción necesaria para hacerlo y en la respuesta de la aplicación ante esas interacciones (experiencia de usuario).

Este orden (que es parte de base del denominado *progressive enhancement*) es importante, porque no podemos permitir que un exceso de carga en la experiencia de usuario ponga en jaque la usabilidad de la aplicación.

Un ejemplo de situaciones en las que estas dos disciplinas a veces no van en sintonía puede ser la propia Apple. Casi se podría afirmar con rotundidad que su dispositivo iPhone dispone de la mejor experiencia de usuario en su mercado; sin embargo, existen muchos procesos relativos al mismo que ponen en serias dudas su usabilidad, como es el caso de transferir archivos al dispositivo desde un ordenador mediante iTunes, frente a la sencillez que pueden aportar otros terminales como los basados en Android.

3.8. Accesibilidad

Según la definición de la Wikipedia:

“La accesibilidad es el grado en el que todas las personas pueden utilizar un objeto, visitar un lugar o acceder a un servicio, independientemente de sus capacidades técnicas, cognitivas o físicas”.

3.8.1. Ideas preconcebidas sobre Flash

A menudo, se suele hacer la rápida amalgama de Flash como tecnología no accesible. Afortunadamente, la realidad no es tan categórica y es bueno saber que Flash Professional (junto con AS3) aporta también herramientas necesarias para hacer aplicaciones más accesibles.

De hecho, ya desde sus inicios, Macromedia presentó un interés especial sobre el tema de la accesibilidad en aplicaciones Flash. De manera que podemos encontrar ya en los *white paper* de Bob Regan del 2005 una serie de buenas prácticas para crear aplicaciones accesibles en Flash (*Best Practices for Accessible Flash Design*). Adobe recogió posteriormente dichas prácticas, a las que ha añadido funcionalidades y paquetes especializados.

A pesar de ello, Flash, por su propia naturaleza Rich Media, posee características multimedia a las que obligatoriamente será necesario dar una alternativa más accesible.

Así, por ejemplo, deberíamos tener una alternativa visual para contenido auditivo (como podrían ser los subtítulos añadidos a un vídeo) o, inversamente, una ayuda audio para contenido visual (como veremos más adelante con el uso de lectores de pantalla para personas de visibilidad reducida).

Esto supondrá, como ocurriría en otra tecnología Rich Media, un trabajo adicional tanto de planificación como de desarrollo.

Por tanto, cuando se realizan proyectos comerciales para empresas privadas, en muchas ocasiones se omiten determinados criterios de accesibilidad por cuestiones más relacionadas con el presupuesto que con las limitaciones tecnológicas, tanto en Flash como en otras tecnologías. Esto no quiere decir que existan determinados puntos básicos que puedan introducirse en prácticamente todas las aplicaciones para mejorar la accesibilidad de la aplicación, sin implicar tampoco grandes partidas presupuestarias.

3.8.2. La accesibilidad como parte integrante del diseño de aplicaciones Rich Media

Antes de entrar en la tecnología Flash propiamente dicha, es importante hacer un inciso sobre el diseño de una aplicación Rich Media orientado a la accesibilidad.

En efecto, **parte de los obstáculos de accesibilidad de una aplicación son fruto de un diseño erróneo** o, mejor dicho, de un diseño sin tener en cuenta esta variable desde el inicio del proyecto. La accesibilidad de una aplicación Rich Media no debe ser un problema a tratar a posteriori, como una capa que se añade al finalizar el desarrollo. No deberíamos crear una aplicación para una mayoría y luego incluir los usuarios discriminados, sino que deberíamos diseñar la aplicación para todos los usuarios posibles desde su etapa inicial. El trabajar a posteriori no solo producirá una aplicación final parcheada, sino que presentará problemas a menudo sin solución, ya que no fueron previstos con anticipación.

Problemas de accesibilidad como podrían ser el tamaño de la letra, colores que dificulten la lectura, interfaces confusas o con elementos de un tamaño excesivamente reducido, etc., son problemas de fácil solución si se tienen en cuenta por todos los perfiles implicados en el proyecto desde un primer momento.

Por otra parte, cabe destacar que cuando hablamos de accesibilidad solemos focalizarnos en discapacidades, pero los problemas de accesibilidad pueden también deberse al dispositivo utilizado. Así, la interacción que podemos tener con un ordenador (monitor, teclado y ratón) no será la misma que la que tengamos con un teléfono móvil (pantalla reducida, sin ratón y con posibilidad de interacción táctil, por teclado o por medios diferentes como un *keypad*, aunque no obligatoriamente).

De la misma forma que sí solemos tratar el dispositivo final a la hora de planificar la aplicación, ¿no sería lógico tener en cuenta todo tipo de usuario, con o sin discapacidades?

Una buena planificación en el diseño de la aplicación nos permitirá solventar buena parte de los problemas de accesibilidad, sea en Flash o cualquier otra tecnología.

3.8.3. Accesibilidad en Flash

Hemos visto que una parte de los obstáculos de accesibilidad son problemas de diseño. Estos problemas podrán solventarse, en general, con una buena planificación de la aplicación y no se verán ligados a la tecnología utilizada.

En cambio, existen otros tipos de obstáculos en los que la tecnología empleada sí es importante: son los casos en que las discapacidades del usuario hacen necesario el uso de tecnologías intermedias, tecnologías de asistencia que le permiten interactuar y comprender el contenido en pantalla. Para ello, Flash debe poder comunicarse con este asistente de manera que el asistente pueda facilitar el contenido de la aplicación al usuario.

Lectores de pantalla en Flash

Uno de estos casos, quizás el más habitual cuando hablamos de accesibilidad, es el uso de lectores de pantalla para usuarios de visibilidad reducida o total. El lector de pantalla (*screen reader*) lee el contenido presente en pantalla. Pero esto no solo implica leer el texto en pantalla, sino la necesidad de que las imágenes, los vídeos o las interfaces de navegación tengan su explicación oral. De esta forma, a través de Flash debemos poder añadir esta información extra y que sea accesible por el lector de pantallas.

Flash, aunque por defecto no hará que la información sea accesible, sí que nos ofrece la posibilidad de activar esta función:

- Por una parte, por medio de la interfaz de Flash Professional podremos añadir información (nombre, descripción, etc.) a la aplicación principal y a sus diferentes elementos, información que puede ser leída por el lector de pantallas.
- Por otra parte, vía AS3, gracias al paquete especializado `flash.accessibility`, podemos asociar información a todo *interactive object* (elemento visible interactivo de una aplicación Flash).

Flash nos permite, de esta forma, no solo añadir información accesible, sino hacer que aparezca en un momento específico, de manera que pueda informar dinámicamente al usuario de cambios en la aplicación (eventos). También po-

dremos comprobar si existe un lector de pantalla activo, de manera a anular animaciones puramente decorativas (como animaciones de texto) que pudieran ser confusas para el usuario.

Desgraciadamente, hay que destacar dos problemas mayores en la compatibilidad de Flash con los lectores de pantalla:

- Flash Player se ha optimizado para Microsoft Active Accessibility (MSAA), de modo que solo será compatible para una serie de lectores de pantalla Windows.
- Añadir información accesible, aunque como hemos visto es posible, requiere también un tiempo de planificación y realización, de forma que habrá que tener en cuenta este tiempo a la hora de realizar la aplicación (con el coste que ello conlleva).

Navegación vía teclado

Otro tipo de discapacidad que se plantea a menudo en la creación de aplicaciones sería el de las dificultades motrices. En general, los usuarios con este tipo de discapacidad preferirán una navegación vía teclado (simplificado) y no vía ratón.

Flash ofrece, para ello, una alternativa parecida a la que encontraríamos en una página web (o sistemas operativos), pudiendo navegar pasando de un elemento a otro con la tecla “Tab” (volviendo al elemento anterior con “Shift+Tab”) y haciendo clic en el objeto que esté enfocado con la tecla “Enter” para seleccionarlo o entrar en él.

Para este tipo de navegación, Flash, en sus objetos de tipo *interactive object*, añade la propiedad `.tabIndex`. Si queremos que un objeto pueda adquirir el foco, le asignaremos un valor a su `.tabIndex`. Estos valores serán enteros, empezando por 1 y prosiguiendo con 2, 3, 4... A cada clic sobre la tecla “Tab” pasaremos a enfocar el elemento que tenga el siguiente `.tabIndex` (una vez llegado al final y si seguimos haciendo clic sobre “Tab”, empezaremos de nuevo en 1).

Crearemos, de esta manera, un orden de tabulación que permite navegar entre los diferentes elementos interactivos de nuestra aplicación sin necesidad del uso del ratón. Flash Player mostrará un rectángulo amarillo sobre el elemento que tenga el foco, a la vez que el lector de pantalla leerá la información de accesibilidad que hayamos puesto previamente.

Enlace relacionado

http://help.adobe.com/es_ES/flash/cs/using/WSd5f4b6c52cfaa4b1-267ca8f5133a966eefa-8000.html#WSd60f23110762d6b883b18f10cb1fe1af6-7c47a.

Observación

En general, muchos usuarios que hagan uso de lectores de pantalla navegarán a su vez vía teclado.

Enlace relacionado

Uso de las propiedades `accessibilityImplementation`, `tabEnabled` y `tabIndex`:
http://help.adobe.com/es_ES/FlashPlatform/reference/actionscript/3/flash/display/InteractiveObject.html.

Ventajas de procurar un contenido accesible

Vemos, pues, que aunque bajo ciertas limitaciones, con trabajo suplementario podremos crear aplicaciones accesibles en Flash. Luego será una cuestión de sopesar los pros y contras de este trabajo extra apenas empezada la concepción de la aplicación.

En este sentido, cabe destacar ciertas ventajas que pueden rentabilizar plenamente el esfuerzo:

- Razones legales (en contenido público en Internet): en ciertos países, cumplir una serie de criterios mínimos de accesibilidad está obligado por ley y el hecho de no cumplirlos puede ser considerado como discriminación en relación con los usuarios.
- Acceso a un mayor público, lo que puede llevar a una rentabilidad mayor de la aplicación.
- En RIA, el hecho de tener una aplicación más accesible permite que sea al mismo tiempo mejor referenciada por los buscadores, de manera a hacerla más visible por usuarios potenciales.

Enlaces relacionados

Centro de recursos de accesibilidad de Adobe:

<http://www.adobe.com/es/accessibility/>.

Fundamentos de la accesibilidad:

<http://www.adobe.com/accessibility/gettingstarted/accessibility.html>.

Creación de contenido accesible con Flash:

http://help.adobe.com/es_ES/flash/cs/using/WSd5f4b6c52cfaa4b1-267ca8f5133a966eefa-8000.html.

<http://mosaic.uoc.edu/ac/le/es/m5/ud1/index.html>.

Manual de mejores prácticas en Flash (2005) en pdf:

http://www.adobe.com/resources/accessibility/best_practices/best_practices_acc_flash.pdf.

Ejemplo práctico y vídeo:

<http://blog.swfjunkie.com/2010/06/flash-accessibility-quest-part1/>.

<http://blog.swfjunkie.com/2010/06/flash-accessibility-quest-part2/>.

4. Codificación

El proceso de codificación hace alusión a la programación de las funcionalidades siguiendo las pautas marcadas por el diseño, de manera que el resultado sea la aplicación a la que el usuario se enfrentará.

En el desarrollo de ARM, la codificación no solamente se ha de centrar en dar una respuesta funcional a las necesidades operativas del usuario (es decir, que permita hacer todo lo que se requiere y que lo haga correctamente), sino que en muchas ocasiones implica el trabajo del lado gráfico y de interacción, lo que la complica de forma adicional frente a aplicaciones en las que la experiencia de usuario e interfaz no tienen tanto peso.

Dentro del desarrollo de ARM, es habitual gastar gran parte de los esfuerzos de codificación en lograr que las transiciones entre elementos en pantalla sean óptimas para trasladar la mencionada experiencia de usuario, o mostrar los tipos de información en pantalla de una forma óptima y de calidad siguiendo las pautas del diseño. Animaciones, transiciones o interacciones son el día a día en las aplicaciones multimedia y no deben entorpecer, en ningún caso, el hecho de lograr que la aplicación cumpla sus funcionalidades básicas.

Es por ello que, en el desarrollo de código, se emplean también técnicas similares a las vistas para el diseño, que permiten hacerse una idea de la funcionalidad básica y de interacción de los diferentes componentes de la aplicación, antes de proceder a darle toda la funcionalidad derivada del diseño, experiencia de usuario, etc.

Entre estas técnicas que permiten agilizar mucho el testeo de funcionalidades de bloques complejos en las aplicaciones, están los **prototipos de desarrollo**.

4.1. Prototipos de desarrollo

A la hora de crear una aplicación, no solo se puede aplicar el concepto de prototipado al diseño, sino que en el área de desarrollo esta técnica puede aportar beneficios interesantes. En todas las aplicaciones con una carga visual y de interacción elevada, es importante poder aislar las funcionalidades de bajo nivel de las funcionalidades derivadas de la interacción y navegación del usuario.

Los prototipos de desarrollo siguen la misma filosofía que existe detrás de los prototipos de diseño, es decir, focalizar los esfuerzos en unos puntos concretos y dejar de lado aspectos más superficiales como transiciones, efectos visua-

les, etc. Esta filosofía es la que fundamenta la técnica denominada *progressive enhancement*, que obtiene su máximo beneficio en el área del diseño web y multimedia aprovechando los ciclos para trabajar sobre la compatibilidad con navegadores, pero que es aplicable a cualquier proceso de desarrollo en el que existan interfaces de usuario e interacciones, como es el caso de las ARM.

Básicamente, el prototipado en desarrollo realizaría un proceso iterativo, partiendo de las funcionalidades críticas de la aplicación (de bajo nivel) para, en sucesivas pasadas, ir aplicando otro tipo de mejoras relacionadas con la interacción, visualización de datos, etc.

Ejemplo

Imaginemos que estamos realizando una aplicación con varias pantallas, una de las cuales trabaja con visualizaciones de datos (cargados de forma masiva desde un servidor) en formato tabulado y en formato de gráfica. Además, el usuario puede insertar determinados valores que modifican los datos obtenidos desde un servidor. El ciclo iterativo sobre los prototipos en desarrollo sería el siguiente:

- Primera iteración con las diferentes pantallas de la aplicación y mecanismo de navegación básica (sin ningún tipo de trabajo visual). En esta pasada, podemos usar un modelo basado en modo texto con cambio brusco y directo entre pantallas.
- Segunda iteración para la pantalla de visualización en la que se realiza la petición al servidor y se muestran los datos tal cual vienen en modo texto. También se enviarían datos a insertar al servidor. De esta forma se podría dar por afianzada la comunicación con el servidor.
- Tercera iteración para dicha pantalla en la que se trabaja la visualización tabulada (sin interacciones complejas) y la visualización en forma de gráfica, aunque sin entrar en el detalle visual.
- Cuarta iteración en la que se dota de funcionalidad adicional a la pantalla, aparecen elementos como un indicador de carga de datos en espera, funcionalidades adicionales para la tabla de datos (ordenación por columnas), etc.
- Quinta iteración en la que se aplicaría el diseño y, si fuera el caso, se programarían efectos visuales y de transiciones en las interacciones con los datos, como por ejemplo mostrar la gráfica de forma animada, etc.

Esta aproximación tiene ventajas no solo a la hora de encontrar errores (al enfrentarse en cada iteración a modificaciones menores y normalmente sobre elementos que ya se sabe que funcionan), sino que además permite controlar el rendimiento en cada iteración para detectar problemas al ir incorporando funcionalidades visuales que, en ningún caso, deben limitar la respuesta básica de la aplicación y que normalmente tienen más opciones para poder ser solventadas.

4.2. Patrones de diseño

En el desarrollo de software también existe el concepto de patrón de diseño, que tiene la misma definición que en el caso de la parte visual, pero con las connotaciones propias de la programación. Los patrones de diseño en desarrollo son soluciones óptimas frente a problemas comunes y que permiten resolver estos problemas de forma elegante y eficiente.

Los patrones de diseño en el desarrollo del software persiguen una serie de objetivos que hacen que no solamente aporten una solución a un problema, sino que lo hagan de una forma que pueda ser extrapolada a situaciones similares futuras:

- Independencia y reusabilidad de la solución.
- Formalización/estandarización de soluciones entre desarrolladores.
- Aprovechamiento del conocimiento previo.

Los patrones de diseño no son un obstáculo frente a la creatividad, sino que tratan de centrarse en los problemas de base y a bajo nivel que se dan en determinadas situaciones del desarrollo. En la actualidad, el uso de patrones de diseño está extendido entre desarrolladores, sobre todo basándose en la programación orientada a objetos, por lo que la lista de patrones de diseño identificados es muy extensa.

En ocasiones, un patrón de diseño tiene diferentes implementaciones debido a los condicionantes del lenguaje en el que se debería programar, pero normalmente se suele contar con alternativas ante estas diferencias.

Un ejemplo de este hecho es la forma de implementar el denominado *Singleton* mediante ActionScript 2 y ActionScript 3. Debido a las diferencias en la definición de algunos elementos de la POO entre ambas versiones del lenguaje, la implementación práctica del patrón varía ligeramente, aunque la filosofía es siempre la misma.

No es el objetivo de esta asignatura adentrarse en esta herramienta de programación, pero hay que ver que hoy en día es fundamental conocer su existencia y entender los grandes beneficios que puede aportar a un programador a la hora de crear software de calidad.

Lecturas recomendadas

Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading: Addison-Wesley.

Eric Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra (2004). *Head First Design Patterns*. Chambersburg: O'Reilly Media.

Joey Lott; Danny Patterson (2006). *Advanced ActionScript 3 with Design Patterns*. Adobe Press.

Ejemplos prácticos:

Dru Kepple (2011). "AS3 101: OOP – Introducing Design Patterns" (<http://active.tutsplus.com/tutorials/actionscript/as3-101-oop-introducing-design-patterns/>).

Eric Feminella (2006). "AS3 Model-View-Controller Implementation" (<http://www.ericfeminella.com/blog/2006/10/08/as3-model-view-controller-implementation/>).

Enlaces relacionados

<http://www.as3dp.com/>.
<http://www.coconnut.com/blog/source/designpatternscard.pdf>.
<http://www.fdi.ucm.es/profesor/jpavon/poo/2.14PDOO.pdf>.

5. Pruebas

La fase de pruebas dentro del ciclo de desarrollo de software tiene especial importancia y, lamentablemente, no suele formar parte de las partidas principales a la hora de definir la planificación del proyecto.

Las pruebas son vitales en el desarrollo de software porque nos permiten **adelantarnos a los problemas** que se podrá encontrar el usuario y detectar fallos que pueden tener connotaciones dramáticas para la aplicación cuando se trata de un servicio. Cada fallo que se detecte antes de que el producto salga a la luz será una frustración y una mala impresión que le estaremos evitando al usuario. Además, cuando la aplicación es un servicio a gran escala, cada fallo que encontremos con antelación podría significar un ahorro importantísimo de desarrollo futuro, una vez que la aplicación ya está en marcha. Imaginemos un error que afecte a la base de datos, en una aplicación que funciona como servicio para miles de usuarios. La solución de ese error tiene o puede tener connotaciones muy diferentes si se detecta en la fase de pruebas, o si se detecta una vez que el producto ya cuenta con esa masa de usuarios y una base de datos con un volumen de información alto.

Si bien todos los desarrolladores realizan pruebas durante el proceso de codificación para tratar de comprobar que las funcionalidades se cumplen siguiendo el guion de los requerimientos de la aplicación, no podemos confiar en este criterio para dar nuestra aplicación por probada. Está comprobado que las personas que están implicadas en el desarrollo trabajan y visualizan la aplicación de una forma no óptima para encontrar errores, debido principalmente a la reiteración de las pruebas a las que se ven sometidos y a su propia percepción de la aplicación al conocer los requerimientos, el trasfondo y los objetivos. A todo esto hay que sumarle el factor de cansancio innato a los seres humanos, que implica a largo plazo (como es el caso de los procesos de desarrollo) dejadez y falta de atención.

Los objetivos de un usuario final no tienen por qué ser los estipulados en los requerimientos (puede darle otra utilidad a la aplicación para la que no fue especialmente concebida), además de que el no conocimiento de cómo está hecho, en muchas ocasiones, implica formas de interactuar diferentes. Estas cuestiones hacen necesario que las pruebas se realicen por **personas ajenas** al proceso productivo de la aplicación y, a ser posible, con perfiles similares a los de los usuarios finales.

Para solventar estas cuestiones, existen **dos tipos de pruebas**:

- **Pruebas unitarias.** Hacen alusión al testeo de funcionalidades concretas, de partes de una aplicación o módulos de la misma. Al poder centrarse en

aspectos concretos, es fácil que los resultados de las mismas terminen por validar que se cumplen todos los requerimientos funcionales necesarios, y que además se cumplen sin errores.

- **Pruebas de integración.** Hacen alusión al testeo de la aplicación en su conjunto, como un todo, e incluso dentro del entorno en el que finalmente será ejecutada dicha aplicación. Una vez que las pruebas unitarias se han completado con éxito, se pueden valorar acciones conjuntas que afecten a más de un proceso o funcionalidad de manera simultánea y que pueden sacar a la luz errores en la intercomunicación entre los diferentes módulos que componen la aplicación.

Para llevar a cabo estas pruebas, se suelen tomar datos de muestra preestablecidos que contemplen diferentes casuísticas en la aplicación. Se pueden determinar conjuntos de datos de prueba que tengan como objetivo comprobar un uso normalizado de la aplicación o, por el contrario, crear conjuntos de prueba erróneos para comprobar que la validación de datos por parte de la aplicación es correcta.

Otro aspecto crítico en las pruebas (sobre todo dependiendo del tipo de aplicación) es el testeo de la **capacidad de carga** de la aplicación o del sistema que la rodea, para asimilar toda la actividad que pueda generar. En ARM basadas en un modelo cliente-servidor, las **pruebas de carga** son fundamentales para asegurar que el sistema completo podrá responder a todas las solicitudes que se gestionen entre los clientes (que pueden actuar de forma simultánea) y el servidor.

Para ello se pueden generar casos de prueba que simulen esta sobrecarga, para determinar los límites de la aplicación o para comprobar que los sistemas de escalabilidad que se hayan podido determinar en la fase de “estrategia y diseño de arquitectura” responden correctamente en situaciones extremas.

Finalmente, hablando de pruebas de un sistema, se podría considerar una **versión beta** como la **prueba definitiva** con un conjunto reducido de usuarios reales. Digamos que la versión beta sería una prueba realizada antes de la puesta en producción, pero ya sobre un entorno real y con todas las funcionalidades probadas anteriormente. Las pruebas en beta se pueden llevar a cabo al final de un desarrollo, o bien se pueden integrar dentro del ciclo de desarrollo del software como un mecanismo de validación de funcionalidades y pruebas unitarias.

No es extraño que grandes compañías incluyan, en su ciclo de desarrollo, a reconocidos usuarios avanzados de sus aplicaciones para que prueben las nuevas funcionalidades que se implementan en futuras versiones, de manera que puedan aportar no solo un reporte de errores encontrados, sino ideas y solicitudes de nuevas funcionalidades a implementar.

6. Instalación y puesta en marcha

Una vez que el software ha sido probado y cumple los requerimientos de calidad que se hayan marcado por parte del cliente y del equipo de desarrollo, se pasa a la instalación y puesta en marcha. Este proceso puede variar notablemente dependiendo de la arquitectura empleada en el desarrollo de la aplicación: desde una aplicación que corre localmente y sin interacción con un servidor, a complejas redes de equipos interconectados.

La fase de instalación y puesta en marcha hace alusión a todos los procesos que se han de llevar a cabo para que el proceso de instalación, por parte de los usuarios, esté accesible y funcionando correctamente, además de que todos los sistemas implicados en el correcto funcionamiento de la aplicación estén correctamente vinculados, configurados y estables.

En el caso de aplicaciones basadas en la nube, esta instalación implica necesariamente el despliegue real de todos los servicios en el lado de servidor, activación de todos los sistemas necesarios de seguridad (tales como copias de seguridad, protección de accesos no autorizados, etc.). Evidentemente, estos mecanismos deberían haber sido desarrollados y testeados previamente en un entorno de pruebas, para poder replicar su activación en esta fase de puesta en marcha.

7. Mantenimiento

El mantenimiento es la fase de desarrollo que permite el **seguimiento**, **corrección** y **evolución** de una aplicación una vez puesta en marcha. El seguimiento servirá para determinar lo eficiente y eficaz que fue el desarrollo, así como detectar nuevas necesidades o errores a solventar.

La corrección de errores perfeccionará la aplicación en base a los problemas que la realidad ha destapado (es muy complicado mediante las pruebas en desarrollo lograr equiparar la intensidad y el volumen de las pruebas a las que se somete una aplicación en un escenario real) y la evolución permitirá mejorarlo en base a nuevas necesidades o funcionalidades que se detecten desde el seguimiento y desde el *feedback* de los usuarios.

Frente a estas tres piezas clave, se definen **cuatro tipos de mantenimiento** diferenciados:

- Correctivos.
- Adaptativos.
- Evolutivos.
- Perfectivos.

El **mantenimiento correctivo** es el más habitual y está destinado a solventar errores detectados en la aplicación, a partir de las funcionalidades inicialmente establecidas.

El **mantenimiento adaptativo** se refiere a las necesidades de cambio que se dan, eventualmente, por modificaciones del entorno en el que la aplicación se desenvuelve.

Ejemplos de este tipo de cambios serían los que tendríamos que realizar en una aplicación que trabaje con la API de Facebook, cuando esta decide modificar sus políticas.

El **mantenimiento evolutivo** hace alusión a las mejoras y cambios que hay que realizar en la aplicación para aportarle nuevas funcionalidades, o incluso eliminar algunas de las existentes si se detectan innecesarias.

El **mantenimiento perfectivo** se basa en la mejora de la aplicación a nivel de código para optimizar procesos o recursos, de manera que la aplicación se perfeccione (no corrigiendo fallos, sino mejorando tareas que ya funcionaban correctamente).