TEMA 1 DESARROLLO DEL SOFTWARE ORIENTADO AL OBJETO

Tema 1
Desarrollo del software orientado al objeto

1. Introducción a la ingeniería del software
2. El OMG y el UML
3. Un ejemplo de aplicación del método Rational Unified Process

1. INTRODUCCIÓN A LA INGENIERÍA DEL SOFTWARE

Un **software** no es una obra de arte, sino un producto de consumo utilitario y masivo; sin embargo es un producto industrial con algunas características especiales: es un producto singular (aunque hay copias de software usados por millones de personas, la producción en serie no nos interesa y no suele ser algo excesivamente generalizado) y no se estropea con el paso del tiempo.

La **ingeniería del software** comprende las técnicas, métodos y herramientas que se utilizan para producirlo. Hoy día la calidad del software y su productividad no han alcanzado un punto óptimo. Por ejemplo, no podemos probar un producto software frente a cualquier posible condición de utilización, lo cual va en contra de su calidad; y en cuanto a la productividad recalcar que cuando diseñamos un software, solemos partir de cero, no como cuando se diseña un coche, donde ya se tienen piezas perfectamente testadas y homologadas (tornillos, chapa, etc) que casi con seguridad sabemos no darán ningún tipo de error en el producto final. Actualmente ya se está intentando conseguir un cierto grado de reutilización de software con el diseño orientado a objetos.

El ciclo de vida del software lo constituyen las etapas que preceden, y las que siguen a la etapa de programación. Existen varios ciclos de vida distintos, con etapas bien definidas, esto es así porque pretendemos cumplir los plazos dados, respetar los límites de los recursos asignados y también ofrecer calidad.

El ciclo de vida clásico

Ciclo de vida clásico

También denominado ciclo de vida en cascada, tiene las siguientes etapas:

- **Análisis previo**: Se definen aquí los grandes rasgos del sistema de software que tendrá que dar soporte informático a unas determinadas actividades. Deberá funcionar en un entorno hardware y red determinado que habrá que indicar; contaremos también con los recursos necesarios para el desarrollo y los condicionamientos temporales. El documento resultante de este análisis es la especificación del sistema y sirve de base para la siguiente etapa.
- **Análisis de requisitos**: Definiremos aquí con detalle las necesidades de información que tendrá que resolver el software, sin tener en cuenta por el momento, los medios técnicos. Deberemos requerir de los futuros usuarios las funciones y requisitos en general; y con esta información redactaremos la especificación de requisitos, con la suficiente precisión para que se pueda desarrollar y servir de base para un contrato entre el equipo de desarrollo del software y sus futuros usuarios.
- **Diseño**: El diseño especifica la solución a cada uno de los requisitos de la fase anterior. Con ello recogemos el documento especificación de diseño, que será la base para la programación.
- **Programación**: Traducimos a código el diseño de la fase anterior.
- **Prueba**: Testeamos el software desde distintos puntos de vista tratando de localizar y corregir en el software y en la documentación todos los posibles errores.
- **Mantenimiento**: o explotación del software, constantes actualización para mejorar la eficiencia o añadir funciones. Este proceso puede durar hasta 10 años o más, por lo que el coste de esta fase suele ser de 2 a 5 veces mayor que el coste de desarrollo.

No obstante, este ciclo ha sido muy criticado porque tiene diferentes inconvenientes; el principal es que los requisitos iniciales que recogemos en el análisis de requisitos muchas veces no son fiables o son incompletos. Es muy difícil encontrar un conjunto de futuros usuarios que conozcan el entorno del software y que sepan exactamente qué es lo que quieres que haga el software. Así que una vez terminada oficialmente la fase de análisis y comenzada la de diseño, seguirán surgiendo nuevos requisitos y cambios; con lo que el software y la documentación pierden validez. Por ello, se han definido otros tipos diferentes de ciclo de vida.

El ciclo de vida con prototipos

Consiste en un software provisional que prioriza la facilidad y rapidez en la modificación sobre la funcionalidad; solo sirve para que los usuarios puedan ver cómo sería el contenido y el posible funcionamiento y apariencia del software, sin realizar realmente estas funciones. Así por comparación, los futuros usuarios podrán definir más exactamente qué es lo que quieren; y por supuesto se puede volver a fabricar otro prototipo y otro más hasta afinar exactamente en el resultado que buscamos.

La programación exploratoria

Se elabora una primera versión del software o de una parte de éste, se le enseña a los futuros usuarios para que la critiquen y se hacen los cambios que estos sugieran, esto se repetirá tantas veces como sea necesario. La diferencia con los prototipos es que en la programación exploratoria, el programa sí funciona y no es un mero prototipo (viene a ser lo que realiza Microsoft con sus betatesters, usuarios a los que les ofrece el programa para que busquen errores y modificaciones, antes de sacarlo a la venta comercialmente en una versión definitiva).

El ciclo de vida del Rational Unified Process

Distingue 4 etapas:

- Inicio: Se establece la justificación económica y el alcance del proyecto.
- Elaboración: Se estudia el alcance del software, a quién debe dar cobertura y se realiza una planificación del proyecto.
- Construcción: Se desarrolla todo el proyecto de forma iterativa e incremental.
- Transición: Se entrega el producto al cliente y comprende todo el comienzo de su utilización , realizando los cambios necesarios y oportunos.

2. EL OMG Y EL UML

El OMG (Object Management Group) se creó en 1989 para fomentar el uso de la tecnología orientada a objetos e impulsar la generación de este tipo de software; es una organización no lucrativa y está constituida por más de 800 empresas.

El UML (Unified Modeling Language) es un modelo para la construcción de software orientado a objetos, que ha sido propuesto como estándar de ISO por el OMG. Con él se ha llegado aun modelo orientado a objetos único de forma oficial, pero eso no quiere decir que todo el proceso en todos los ingenieros sean únicos. De momento se ha conseguido que haya unos diagramas que todos los desarrolladores entenderán y harán de la misma manera; lo cual supone un importante avance, pero no es definitivo.

3. UN EJEMPLO DE APLICACIÓN DEL MÉTODO RATIONAL UNIFIED PROCESS

3.1. Documentación de requisitos

Los **requisitos** son la especificación de lo que debe hacer el software, son descripciones del comportamiento, propiedades y restricciones del software que debemos desarrollar. El papel de los requisitos es el de servir de base para un acuerdo entre los usuarios y los desarrolladores del software (en este sentido debe estar realizado de una manera inteligible para los usuarios que deben revisarlo) y además son la información de partida para desarrollar el software, es decir, son el pie que da entrada a la siguiente etapa.

Hay dos **clases de requisitos**: Los requisitos funcionales describen qué debe realizar el software para sus usuarios, mientras que los no funcionales no van asociados a casos de uso en concreto y son restricciones impuestas por el entorno y la tecnología.

Las **fuentes de información** a las que debemos recurrir para recoger la información sobre los requisitos son:

- **Entrevistas** y eventualmente **encuestas** a los futuros usuarios, si además podemos observarles directamente en su puesto de trabajo, mucho mejor.
- **Documentación sobre el sistema actual** existente, y si está informatizado también echaremos mano de los manuales y procedimientos.
- **Colegas de los usuarios** para conocer el mismo trabajo desde otros puntos de vista.
- **Revisión de sistemas parecidos** dado que el usuario no mencionará funciones importantes del software que para él no son visibles o no le da la debida importancia.

Los pasos de la recogida y documentación de requisitos son:

1. El **contexto del software**: Se trata de recoger los requisitos conociendo e indagando en la actividad profesional de los usuarios. Informáticamente sabemos trabajar, pero no conocemos el entorno organizativo que nos piden para ese software. A su vez existen dos modelos para describir este contexto. El **modelo del dominio** es la más simplificada y recoge los tipos de objetos más importantes para establecer una clasificación. Un modelo más complejo es el **modelo de negocio** que describe los procesos y entidades principales entorno al software. En éste último modelo se profundiza identificando todos los casos de uso, las entidades que participan en él…
2. Los **guiones**: Se denomina así a lo que los usuarios conocerán como casos de uso, es decir, identifican de manera organizativa lo que quieren que haga el software; son una fuente de información muy importante, ya que expresan las necesidades de los usuarios tal como ellos las ven.
3. **Identificación de los actores**: Distinguimos principalmente el usuario final (colectivo o personas que interactuarán directamente con el sistema), el usuario privilegiado (o gestor del sistema, encargado de definirla forma de funcionamiento del sistema) y el entorno informático (interfaz no humana del software).
4. **Identificación de los casos de uso**: Se obtienen a partir de los guiones, y son procesos autónomos iniciados por un actor o por otro caso de uso.
5. **Identificación de las relaciones entre casos de uso**: Ya sabemos que hay 3 tipos de relaciones principales: las relaciones de extensión (siempre se relacionan con una condición, como pueden ser diferentes flujos de proceso o casos especiales o errores que debemos tratar); las relaciones de inclusión (es un recurso para evitar la descripción de una misma parte del proceso dentro de diferentes casos de uso); y la relación de especialización (un caso de uso es versión especializada de la otra).
6. **Identificación de las relaciones de especialización entre actores**: El actor A e suna especialización del B si A tiene todos los apeles de B y alguno más. Esto no suele representar problemas para identificarlo.
7. **Documentación de los casos de uso**: Encontramos dos tipos de documentación: la textual y la formal. En la **documentación textual** encontramos la descripción textual donde aparece el nombre de los actores,

otra terminología que se pueda usar con ellos y referencias a otros casos; y también contiene un glosario, muy conveniente para unificar terminología y su interpretación. Por otro lato también está la **documentación formal** que consiste en un diagrama de los casos de uso que muestre todas las relaciones posibles entre éstos y entre los actores; pero no entramos en demasiada profundidad, es decir, solo los utilizamos como función complementaria y no se elaboran sistemáticamente para todos los casos de uso. Más adelante, utilizaremos los diagramas de análisis en esta fase de análisis con mayor profundidad.

3.2. Análisis

El análisis tiene 3 principales cometidos:

- Traducir los requisitos a un lenguaje más formal. En la etapa anterior de recogida de documentación y requisitos se han descrito las funciones del software en forma de casos de uso y de tareas, ahora bien, un desarrollo posterior orientado a objetos debe utilizar un formalismo de clases y objetos que todavía no tenemos.
- Identificar las clases fundamentales para la implementación del software.
- Expresar los casos de uso en términos de clases.

Existe un alto grado de continuidad entre el análisis y la etapa posterior de diseño, tal es así que probablemente la mayoría de las clases identificadas en la etapa de análisis serán ya definitivas en el diseño posterior. La utilidad del análisis por lo tanto es fundamental pues se puede analizar todo el software con un coste relativamente bajo; hay veces que no se desarrolla un modelo de análisis, es decir, de los requisitos se pasa directamente al diseño, pero solo es recomendable en casos extremadamente sencillos.

Por sencillo que sea un proyecto de software, es conveniente dividir la documentación en **paquetes de UML**. Cada paquete debe ser coherente (los elementos que constituyen el paquete deben estar fuertemente relacionados) y poco dependiente (pocas conexiones entre elementos de paquetes diferentes). A su vez este paquete de análisis se divide en paquetes de servicio, es decir, por su funcionalidad y cada paquete de servicio sólo puede estar en un paquete de análisis a la vez.

Los paquetes de análisis corresponden cada uno de ellos a uno o varios subsistemas enteros y pueden servir para repartir el trabajo del análisis entre varias personas o equipos. Normalmente, los casos de uso entre los que hay relaciones de extensión, inclusión o generalización deben asignarse al mismo paquete.

Los paquetes de servicios son subdivisiones de los paquetes de análisis desde un punto de vista que podríamos llamar *comercial*. Comprenden normalmente casos de uso relacionados y habitualmente tienen que ver con un único actor o, en cualquier caso, con pocos; son indivisibles (o un cliente tiene un paquete de servicios completo, o no lo tiene).

Hay veces que es necesaria una **revisión de los casos de uso**, especialmente si en la fase anterior solo se implementaron estos casos de uso para servir de acuerdo entre usuarios y desarrolladores. Si no se han estudiado a fondo estos casos, puede ocurrir que haya dos casos que realicen funciones parecidas quizá de forma contradictoria.

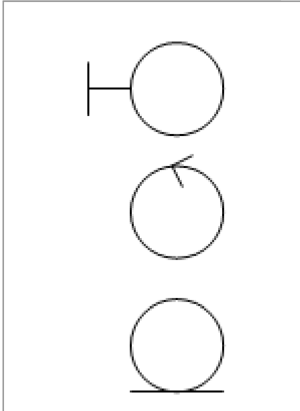
Se consideran **tres tipos de clases de análisis**:

- **Clases de frontera**: Representan en el nivel de análisis la interfaz de usuario por pantalla. Debe haber al menos una para cada papel de cada actor y representan objetos gráficos complejos como ventanas, diálogos de pantalla,

menús… Si modificamos en el futuro la forma de presentar en pantalla los datos pero sin afectar al proceso de los mismos, solo habremos de cambiar las clases frontera.

- **Clases de entidades**: Corresponden a los objetos del dominio que formalmente serán persistentes, es decir, durarán más que el proceso que los crea.
- **Clases de control**: Corresponde a objetos internos del software, no persistentes, no modelan nada del mundo real y sus operaciones suelen contener la parte principal de los algoritmos de aplicación. Si en el futuro tenemos que variar los algoritmos de proceso sin cambiar nada en los datos, solo será necesario modificar las clases de control.

Clases de análisis: frontera, control y entidad



Uno de los problemas que se da es el **identificar las clases de entidades**, es importante hacerlo correctamente porque la mayoría pasarán a la fase de diseño, existen algunas formas de identificarlas, pero no hay una regla maestra, por ejemplo: bibliotecas de clases ya definidas y programadas que además podremos reutilizar; clases sugeridas por los expertos en el dominio, la documentación revisada de los casos de uso…

Normalmente rechazaremos las clases sin atributos, las que no tengan operaciones, las que solo tengan un atributo (ya que pueden ser realmente el atributo de otra) y las clases con un solo objeto.

3.3. Diseño

Recordemos que siguiendo el ciclo de vida del Rational Unified Process, el análisis y el diseño constituyen un único componente de proceso, pero nosotros los consideraremos dos etapas diferentes porque tienen una finalidad distinta y porque el resultado también es claramente diferente.

La etapa del diseño hace de puente entre el análisis yla realización. El modelo del análisis describe las funciones que tiene que desempeñar el software (plantea el problema) mientras que las etapas siguientes deben buscar las soluciones a esas especificaciones.

Hemos visto que en proyectos muy sencillos se puede llegar a prescindir del análisis. En cambio, el diseño es siempre necesario.

La **reutilización** es fundamental en el diseño del software y por sí misma, es ya motivo más que suficiente para justificar la existencia de esta fase. Encontramos que con una buena reutilización conseguimos varias ventajas:

- Se abrevia el desarrollo del software, disminuye el trabajo de mantenimiento y mejoramos la fiabilidad.
- El código reutilizado suele ser más eficiente porque se ha ido mejorando con el tiempo.
- Se gana en coherencia: dado que se suelen reutilizar varias clases y por ello suelen estar programadas de la misma forma y con el mismo estilo.
- Si se reutiliza un buen fragmento de código, será mucho más difícil que se pierda ya que habrá muchas más copias.

La reutilización tiene 4 modalidades: la de clases, de componentes, los patrones y los marcos. La **reutilización de clases** suele acontener cuando éstas son independientes del dominio como interfaces gráficas y estructuras de datos generales. La **reutilización de componentes** requiere primero saber qué es un componente: Es un conjunto de clases cuyos objetos colaboran en tiempo de ejecución con el fin de llevar a cabo una función concreta. Dado que un componente implementa una interfaz determinada (todas las operaciones de sus clases que se pueden pedir del exterior) se puede sustituir un componente por otro que implemente la misma interfaz.

Los **patrones** son una manera organizada de recoger la experiencia de los diseñadores de software para volverla a utilizar en casos parecidos. Cuando un experto informático inventa una solución, rara vez la invención es completamente suya, sino que adapta parte de una solución que ya existía. Esto son los patrones, no son en sí programas sino ideas de diseño extraidas de la experiencia que constituyen un esbozo de solución o receta para problemas parecidos que pueden aparecer con frecuencia, y que solo debemos adaptar a nuestro caso particular.

Así la reutilización avanza un paso más; en casos determinados en los que no se puede reutilizar código, al menos se puede reutilizar el diseño, como mínimo, las ideas básicas. Ya existen "recetarios" de patrones preexistentes que, además, van aumentando con el uso de sí mismos, así encontramos que patrones nuevos ya hacen referencia a patrones existentes anteriormente a ellos.

Las **características de los patrones** son:

- Recogen la experiencia, es decir, no se inventan completamente de nuevo.
- Son algo más amplio que una clase u objeto.
- Crean vocabulario: dentro de un diseño se hará referencia a los patrones por su nombre; por ello el nombre que le asignemos al patrón y por el cual se conocerá no debe ser trivial.
- Son un instrumento de documentación, ya que hacer referencia un patrón dentro de nuestra documentación nos ahorrará describirlo enteramente.
- Dan una solución genérica a un caso concreto, que nosotros debemos completar.

Los **componentes de los patrones** son:

- **Nombre**: Ya vimos que no es trivial, en los patrones no puede haber sinónimos o alias.
- **Contexto**: El entorno dentro del cual se presenta el problema que es resuelto por el patrón.
- **Problema**: Requisitos que la solución debe cumplir.
- **Solución**: El patrón.

Los patrones, a menudo, hacen referencia a otros patrones, ya sea porque los utilizan o porque los complementan. Cuando utilizamos un sistema de patrones (bastante frecuente) hay que documentar las relaciones entre éstos y la descripción de las dependencias entre ellos cuando se utilizan juntos.

Los **marcos de aplicaciones** son un conjunto de clases que constituyen una aplicación incompleta y genérica. Si el marco se complementa de manera adecuada (se especializa) se obtienen aplicaciones especializadas de un cierto tipo. Hay dos tipos de marcos:

- Marcos de caja blanca: Con un conjunto de clases de las cuales está definida la interfaz y la implementación. Para especializarlo hay que implementar las subclases de estas clases.
- Marcos de caja negra: Tienen definidas unas interfaces denominadas papeles, y para especializarlo hay que añadirles clases que implementen sus papeles.

Los marcos tienen varias ventajas: Reducen el trabajo al programar, dado que aplicamos subsistemas que sabemos que funcionan y por ello el código no se deberá sobrescribir ni mantener; llevan a desarrollar pequeñas aplicaciones que encajan dentro de los marcos, en lugar de aplicaciones monolíticas; los marcos permiten que otras compañías puedan suministrar componentes que los desarrolladores podrán añadir. Pero a su vez, también encontramos inconvenientes: Limitan la flexibilidad pues los componentes para un marco deben amoldarse a las restricciones impuestas por la arquitectura de éste; tienen un aprendizaje difícil, como media se requieren 3 semanas para estudiarlo a fondo, aunque ciertamente solo se debe realizar una vez y puede servir este aprendizaje para muchas aplicaciones que se basen en este marco estudiado; reducen el grado de creatividad de los desarrolladores.

Aunque los marcos y los patrones pudieran parecernos lo mismo, son bastante diferentes:

- Los patrones son soluciones más pequeñas que un marco: un marco puede contener varios patrones, pero no al revés.
- Los patrones son más abstractos.
- Los patrones son menos especializados, ya que se pueden utilizar en cualquier tipo de aplicación.

El **diseño arquitectónico** tiene como objetivo definir las grandes líneas del modelo del diseño; y comprende las actividades siguientes:

- **Establecimiento de la configuración de la red**: Es decir, determinar los nodos y sus características, las conexiones entre ellos, ancho de banda…
- **Establecimiento de los subsistemas**: Pueden ser subsistemas propios o desarrollados por otras compañías o incluso software de mercado. Podemos también aplicar patrones arquitectónicos y generalmente aplicaremos la división de paquetes realizada en la etapa de análisis. Las dependencias entre subsistemas serán como mínimo, las dependencias que ya existen entre los paquetes de análisis y los paquetes de servicios correspondientes.

El **diseño de la implementación de los casos de uso** parte del diagrama de colaboración resumido que se ha hecho en la etapa de análisis, y se consideran por separado las clases de frontera, de entidades y de control.

Las clases de frontera son el punto de partida del diseño de la interfaz de usuario. Las clases de control y de entidades sirven para implementar la funcionalidad de los casos de uso. Debemos intentar aprovechar la oportunidad de reutilizar componentes y aplicar patrones y marcos siempre que sea posible.

El proceso de diseño de las clases de control comienza con el estudio de la implementación de las operaciones ya identificadas en el análisis, una por una. A menudo necesitaremos nuevas operaciones de clases para implementar éstas o incluso clases nuevas. Después habrá que estudiar la implementación de estas nuevas operaciones. Con las nuevas clases y operaciones llegamos al **diagrama estático de diseño**.

Generalmente el diagrama estático de diseño se va haciendo a la par que se diseñan los casos de uso, y una vez acabado ello, debemos hacer una revisión del diagrama obtenido. La revisión del diagrama obtenido (que suele contener 5 veces más clases que el diagrama de análisis) se basa en:

- **Normalización de los nombres**: Por continuidad será bueno utilizar la misma terminología del diagrama estático del análisis. Aunque es posible que debamos cambiar algunos nombres bien porque vulnerar alguna norma aplicable al proyecto, bien para respetar una terminología ya establecida en proyectos anteriores o bien porque el lenguaje de programación no lo soporta (esto último debíamos haberlo previsto con anterioridad.

- **Reutilización de las clases**: Durante el diseño las clases se hacen a medida, ahora se trata de revisarlas sistemáticamente en busca de clases que pueden ser reutilizadas por otras ya existentes.

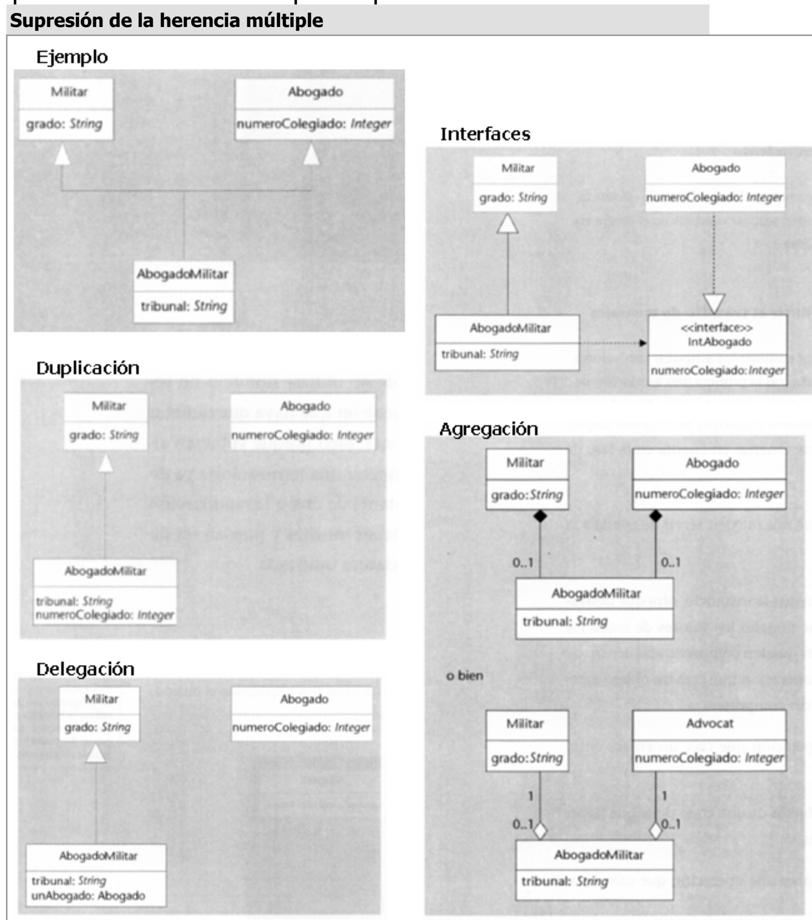
- **Adaptación de la herencia en el nivel soportado por el lenguaje de programación**: La herencia múltiple no suele ser soportada por todos los lenguajes. Por ello existen varias formas de deshacerla: por **duplicación** consistente en duplicar en la subclase los atributos que heredaría de una de las superclases; por **delegación**, manteniendo una sola de las superclases e incluyendo dentro de la subclase una referencia a un objeto de la otra superclase, que tiene el valor de los atributos correspondientes; por **interfaces**, sustituyendo la herencia doble por herencia simple más una

interfaz implementada por otra superclase; o por **agregación**, como podemos observar en la imagen de la página siguiente.

- **Sustitución de las interfaces**: Si el lenguaje de programación no soporta las interfaces, las sustituiremos por clases abstractas.

- **Cambios para la mejora del rendimiento**: Ahora en el diseño nos debemos preocupar por cuestiones de eficiencia que en el diseño no nos preocupaban.
- **Agrupación de clases para reducir el tránsito de mensajes**: Antes habíamos sustituido los atributos con valores múltiples por una clase aparte que comparta con la primera una asociación de n a 1; pues bien, quizá ahora nos interese el cambio a la inversa, ya que así no hay que enviar un mensaje de una clase a otra.

Supresión de la herencia múltiple



- **Especificación de las operaciones implícitas**: En general para toda clase debe haber una operación que la instancia, otra que destruya objetos y operaciones que lean y pongan los valores de todos los atributos.

- **Referencias a las clases frontera**: Hay que sustituir las operaciones de las clases de frontera por operaciones en elementos de la interfaz de usuario.

- **Cohesión y acoplamiento**: Ya estamos casi seguro de tener un diagrama estático con todas las clases que deberá tener para sus implementaciones, y con todas sus operaciones bien definidas. Pues es ahora cuando hay que revisarlo para que sea **coherente**; es decir, los atributos y operaciones de cada clase deben tener mucha relación entre sí, hasta el punto de considerarla una unidad. Las clases y operaciones poco coherentes son más difíciles de entender, presuponen relacionar varias entidades en relación de 1 a 1, y se ven afectadas por más modificaciones. El **acoplamiento** también es importante y expresa el grado en el que éstos dependen de otras clases y objetos para llevar a cabo su responsabilidad; es mejor un menor acoplamiento, dado que cuanto más tenga una clase, más se vera afectada por cambios en otras y por tanto, se deberá modificar más a menudo.

Se llama **clases persistentes** a las que pueden tener objetos persistentes, y clases temporales a las no persistentes. Un objeto persistente es aquel que tiene una vida más larga que el proceso que lo crea, o dicho de otra forma, un objeto se crea en un proceso y se utiliza en procesos posteriores, por lo que hay que grabarlo en un sistema de almacenamiento permanente y también lógicamente debe poder ser leído. Existen tres tipos principales de sistemas de almacenamiento: bases de datos orientadas a objetos, bases de datos relacionales y ficheros clásicos; y base de datos object-relacional.

Bases de datos orientadas a objetos: Como es lógico suponer es el caso más sencillo de todos, ya que no hay que transformar los objetos para hacerlos persistentes. Solo hay que enriquecer la definición de la clase con objetos

persistentes, indicando cuales de sus atributos lo son y qué política de lectura de objetos se requiere: todos de una vez, o leer según demanda).

Bases de datos relacionales y ficheros clásicos: Aquí la cuestión ya se complica un poco: a un objeto le corresponde en principio, una fila de una tabla de base de datos relacional o un registro de un fichero; y a los atributos le corresponden columnas de la tabla correspondiente. Se hace necesario realizar la transformación anterior antes de grabar un objeto y hay que llevar a cabo la inversa antes de poderlo utilizar para ser leído. Existen tres formas de llevar a cabo la gestión de la persistencia:

- Cada clase persistente tiene operaciones para que los objetos se graben, borren, etc, por sí mismos. Es la opción más eficiente dado que requiere menos llamadas a objetos, pero dependerá de cada gestión de base de datos concreta y, por tanto, no es una solución portable a otro sistema distinto.
- Gestor de disco para cada clase persistente: Este gestor accede directamente en cada clase al fichero o base de datos, así la clase de entidades se desacopla del sistema de gestión de base de datos y además el gestor puede servir de memoria caché intermedia. Esta solución es menos eficiente que la anterior.
- Mezcla de las dos anteriores: se crean gestores de disco y se añaden operación de grabación, lectura en cada clase de dominio. Se diferencia del primer método en que puede ser más portable, sobre todo si implementamos la persistencia por medio de un marco.

Supresión de la herencia

CASO A
Tabla Cliente
Nif: string, clave principal
Nombre: String
....
Tabla ClienteEspecial
NIF: String, clave principal
Nombre: String
Descuento: Integer

CASO B
Tabla Cliente
Nif: string, clave principal
Nombre: String
....
Tabla Auxiliar ClienteEspecial
NIF: String, clave principal
Descuento: Integer

CASO C
Tabla Cliente
Nif: string, clave principal
Tipo: Integer
Nombre: String
....
Descuento: Integer
...

En cualquiera de los casos debemos pasar por 4 fases para **definir la estructura** de base de datos relacional o con ficheros clásicos:

1. Transformamos el modelo estático en un modelo entidad-relación: se convierte cada clase con un atributo con valores múltiples en dos clases unidas por una asociación, y después le hacemos corresponder un tipo de entidad a cada clase no asociativa. Después se añade una relación para cada asociación con cardinalidades que son obvias.
2. Supresión de la herencia: Como el sistema de base de datos no soporta la herencia podemos resolverlo:
   a. Definiendo una tabla para cada clase que contendrá los atributos propios de la clase y los heredados. Es una solución sencilla pero tiene tantas tablas como subclases, hay que crear un gestor de disco para cada subclase, un cambio en la definición de "cliente" obligaría a cambiar todos los gestores de disco…
   b. Se puede crear una tabla para la superclase en la que se graba a todos los clientes; a la vez se crea una tabla complementaria para cada subclase con el identificador del objeto y los atributos específicos (en este caso el descuento). Es buena solución si sólo una pequeña fracción de los objetos de la clase pertenecen a la subclase y además si los valores de los atributos son de longitud fija.
   c. Se crea una única tabla para todas las jerarquías de herencia con todos los campos, tanto de la superclase como de todas sus subclases y se le añade un campo que nos diga qué subclase del objeto corresponde a cada fila. La eficiencia en tiempo es muy elevada en esta solución, pero menos en ocupación de disco.

El **gestor de disco** es una clase diferente de la clase persistente y, dentro de las operaciones de ésta, hay llamadas a operaciones del gestor de disco. Podemos considerar que todos los gestores de disco tienen al menos unas operaciones básicas: leer, grabar, regrabar y borrar. Por ello todos los gestores son subclases de una superclase GestorGenerico, en la cual las operaciones serían abstractas.

El diseñar **gestores de disco para la materialización según demanda** (lectura según demanda) es más compleja y sólo se justifica cuando la

materialización es costosa o cuando se debe poder sustituir la clase de entidad sin tener que modificar todas las que la utilizan. En resumen las referencias exteriores de estos gestores no se hacen a objetos de la clase de entidades en cuestión, sino a objetos de una clase sustituta. La clase original y la sustituta implementan la misma interfaz.

Bases de datos object-relational: Es el mismo diseño que las bases de datos relacionales, con la diferencia de que este nievo modelo puede tener atributos de tipo compuesto y, por tanto, no será necesario crear gestores de disco para todas las clases persistentes, sino sólo para aquellas a las cuales se accederá directamente.

TEMA 2 PATRONES DE DISEÑO

Tema 2
Patrones de diseño
1. La reutilización en el desarrollo orientado al objeto
2. El concepto de patrón
3. Una selección de patrones

1. LA REUTILIZACIÓN EN EL DESARROLLO ORIENTADO AL OBJETO

Ya sabemos que la principal ventaja del software orientado a objetos es la reutilización, que tiene cuatro modalidades: reutilización de clases, de componentes, de patrones y las construcciones.

Con la reutilización se abrevia el desarrollo del software, se disminuye el trabajo de mantenimiento del software en el futuro, se mejora la fiabilidad, el código es más eficiente y coherente y se preserva la inversión.

2. EL CONCEPTO DE PATRÓN

Los **patrones** son una manera organizada de recoger la experiencia de los diseñadores de software para volverla a utilizar en casos parecidos. Cuando un experto informático inventa una solución, rara vez la invención es completamente suya, sino que adapta parte de una solución que ya existía. Esto son los patrones, no son en sí programas sino ideas de diseño extraidas de la experiencia que constituyen un esbozo de solución o receta para problemas parecidos que pueden aparecer con frecuencia, y que solo debemos adaptar a nuestro caso particular.

Así la reutilización avanza un paso más; en casos determinados en los que no se puede reutilizar código, al menos se puede reutilizar el diseño, como mínimo, las ideas básicas. Ya existen "recetarios" de patrones preexistentes que, además, van aumentando con el uso de sí mismos, así encontramos que patrones nuevos ya hacen referencia a patrones existentes anteriormente a ellos.

Las **características de los patrones** son:

- Recogen la experiencia, es decir, no se inventan completamente de nuevo.
- Son algo más amplio que una clase u objeto.
- Crean vocabulario: dentro de un diseño se hará referencia a los patrones por su nombre; por ello el nombre que le asignemos al patrón y por el cual se conocerá no debe ser trivial.
- Son un instrumento de documentación, ya que hacer referencia un patrón dentro de nuestra documentación nos ahorrará describirlo enteramente.
- Dan una solución genérica a un caso concreto, que nosotros debemos completar.

Los **componentes de los patrones** son:

- **Nombre**: Ya vimos que no es trivial, en los patrones no puede haber sinónimos o alias.
- **Contexto**: El entorno dentro del cual se presenta el problema que es resuelto por el patrón.
- **Problema**: Requisitos que la solución debe cumplir.
- **Solución**: El patrón.

Los patrones, a menudo, hacen referencia a otros patrones, ya sea porque los utilizan o porque los complementan. Cuando utilizamos un sistema de patrones (bastante frecuente) hay que documentar las relaciones entre éstos y la descripción de las dependencias entre ellos cuando se utilizan juntos.

Los patrones se **clasifican** en:

- **Patrones de notación y diagramas**: Se pueden aplicar a la hora de hacer determinados diagramas, como los de clases y sirven tanto para el análisis como el diseño; en principio eran independientes del modelo utilizado, pero debido a la aceptación del UML suelen ser específicos de este modelo.
- **Patrones de análisis**: Pueden ser generales o específicos, específicos por ejemplo de un modelo como UML o de un dominio, como el financiero.

- **Patrones de diseño**: Distinguimos aquí los patrones arquitectónicos (estructuración de un sistema en subsistemas), de diseño (los propiamente dedicados a resolver problemas de diseño) o frases hechas (que describen la manera de implementar algo en un lenguaje de programación concreto).

Los **sistemas de patrones** se llaman precisamente sistemas porque a menudo un patrón hace referencia a otro u otros, y es entonces cuando todos deben cumplir determinadas características comunes:

- El sistema debe contener un número suficiente de patrones de todos los tipos anteriores, para poder hacer un diseño basado en patrones
- Estos patrones deben estar descritos de una manera lo más uniforme posible.
- Las relaciones entre patrones deben estar indicadas explícitamente.
- El catálogo debe estar bien organizado según diferentes criterios, para poder encontrar el que necesitemos en un momento dado con facilidad.

Para **seleccionar el patrón adecuado** debemos seguir los siguientes pasos:

1. Especificar por escrito el problema que intentamos resolver con los patrones, descomponerlo en subproblemas y describir las fuerzas que afectan a cada uno de ellos.
2. Delimitar el conjunto inicialmente de patrones aplicables mediante el catálogo.
3. Afinar la selección son un estudio posterior más pormenorizado.
4. Añadir a la selección del punto anterior todos los patrones relacionados que pudiesen existir.
5. Considerar las ventajas, inconvenientes y limitaciones de cada patrón elegido.
6. Seleccionar la variante más adecuada (de existir) de los patrones elegidos.

Una vez seleccionado, para **aplicar** adecuadamente el patrón:

- Haremos una lectura general de toda la documentación del mismo.
- Estudiaremos todos los apartados de estructura y participante.
- Estudiaremos los ejemplos resueltos si los hay.
- Compararemos la terminología abstracta que el patrón nos ofrece con el problema real que queremos resolver.
- Definiremos las nuevas clases necesarias y modificaremos las existentes para adaptarlas a nuestro problema.

---

3. UNA SELECCIÓN DE PATRONES

Patrones de arquitectura

Broker

- Tema: Cliente/Servidor
- Contexto: Un sistema de software distribuido.
- Problema: Coordinar la comunicación entre componentes independientes.
- Ventajas, inconvenientes y limitaciones: Separa los clientes de los servidores, así que ambos pueden cambiar de lugar sin verse afectados; se pueden añadir tantos clientes como servidores; pero el hecho de que se trate de una arquitectura distribuida hace que sea menos eficiente por las llamada intermedias y puede fallar por más puntos que una centralizada.

Layers

- Tema: Responsabilidad
- Contexto: Sistema de software complejo.
- Problema: Descomponer el software de una manera adecuada desde el punto de vista de la portabilidad y la posibilidad de sustituir algunas partes.

- Ventajas, inconvenientes y limitaciones: Los distintos niveles se pueden reutilizar por separado, un software con niveles se mantiene mejor, las interfaces se pueden estandarizar.

Model-View-Controller
- Tema: Responsabilidad
- Contexto: Software interactivo.
- Problema: Separar la función principal del tratamiento de las entradas por un lado, y de las salidas por el otro, que se pueden tratar por medio de herramientas diferentas o se puede cambiar la forma.
- Ventajas, inconvenientes y limitaciones: De un solo modelo puede haber diferentes vistas, pudiéndose sustituir sin afectar al modelo (aunque los cambios en el modelo sí pueden afectar a las vistas); se garantiza la propagación de los cambios en el modelo a vistas y controles; el problema es que inicialmente se concibió para la programación en Smaltalk, no asegurándose que sea igual de adecuado en otros lenguajes orientado a objetos.

Pipe and filters
- Tema: Responsabilidad
- Contexto: Hay una corriente de datos de entrada que se tiene que procesar en pasos sucesivos.
- Problema: Poder cambiar el orden de los pasos, permitir la ejecución de pasos en paralelo, coger los datos de entrada de diversas fuentas y representar los datos de salida de diversas maneras.
- Ventajas, inconvenientes y limitaciones: Cuando se cambian los filtros, se cambian los pasos y también se modifica el proceso; se evita la necesidad de crear ficheros para pasar la información de un paso al siguiente; los filtros son reutilizables, permite el procesamiento en paralelo, aunque el tratamiento de los errores es complicado porque no hay un estado de ámbito global.

Presentation-Abstraction-Control
- Tema: Responsabilidad
- Contexto: Una aplicación interactiva.
- Problema: La aplicación se estructura como un conjunto de agentes, cada uno de los cuales tiene los datos y la interfaz de usuario, los agentes están sujetos a cambios principalmente en la interfaz de usuario y se pretende que los cambios en un agente no afecten a los otros.
- Ventajas, inconvenientes y limitaciones: Permite la especialización de la presentación y la abstracción; los cambios en las mismas de un agente no afectan a ningún otro agente, permite el procesamiento en paralelo ya que los agentes se pueden asignar a procesadores, ordenadores, tareas o hilos de ejecución diferente. Eso sí, la descomposición en agentes, aumenta la complejidad de la aplicación y disminuye la eficacia cuantos más agentes hay, por lo que hay que buscar un número óptimo de los mismos: no es conveniente cuando hay muchos agentes pequeños con interfaces de usuario análogas.

Adapter
- Tema: Interfaz
- Contexto: Reutilización de clases.
- Problema: Se quiere reutilizar una clase, pero su interfaz no lo permite.
- Ventajas, inconvenientes y limitaciones: Puede servir tanto para cambiar sólo los nombres de las operaciones como para sustituir un conjunto de operaciones por otro, una clase adaptada tiene más posibilidades de reutilización.

Bridge
- Tema: Interfaz
- Contexto: Una interfaz que puede tener diversas implementaciones alternativas por medio de sendas clases.

- Problema: Se quiere evitar que si se define una subclase de la interfaz se tengan que implementar tantas subclases como clases implementan la interfaz.

Builder
- Tema: Instanciación
- Contexto: Objetos Compuestos.
- Problema: Hacer que la construcción de un objeto compuesto sea independiente de su representación, de manera que el proceso de construcción no se vea afectado por el hecho de que cambie la forma de representación.
- Ventajas, inconvenientes y limitaciones: Hace más fácil cambiar la forma de representación del objeto compuesto, pero no cambia la estructura; la construcción se realiza paso a paso, cosa que facilita el control del proceso.

Controller
- Tema: Responsabilidad
- Contexto: Mensajes de acontecimientos del sistema.
- Problema: Hay que decidir qué clase es responsable de tratar los mensajes de acontecimientos del sistema.
- Ventajas, inconvenientes y limitaciones: Como la misma clase trata todos los acontecimientos que tienen repercusiones en el caso de uso, se puede detectar fácilmente si alguno llega fuera de secuencia; solo puede ser conveniente que haya un único controlador para todo el sistema cuando se vayan a tratar pocos acontecimientos.

Creator
- Tema: Instanciación/Responsabilidad
- Contexto: Instanciación.
- Problema: Hay que decidir qué clase es responsable de crear los objetos de una clase de A (por ejemplo permite añadir a un objeto que graba un fichero una operación que comprima el fichero antes de grabarlo).
- Ventajas, inconvenientes y limitaciones: El hecho de que se encargue la creación de objetos de A a una clase que ya depende de éstos por otras razones, evita el acoplamiento innecesario de otra clase de A.

Decorator
- Tema: Interfaz
- Contexto: Los objetos de una clase pueden tener que hacer muchas operaciones diferentes, muchas de las cuales o bien se hacen raramente, o bien no se conocen hasta la hora de la ejecución.
- Problema: Añadir operaciones y atributos a un objeto en tiempo de ejecución.
- Ventajas, inconvenientes y limitaciones: En la definición de la clase podemos poner las operaciones "regulares", mientras las especiales aparte; si proliferan los objetos decoradores, el diseño pierde claridad; un objeto y sus decoradores no tienen la misma identidad lo cual limita la posibilidad de utilizar técnicas basadas en ésta.

Don´t talk to strangers
- Tema: Responsabilidad
- Contexto: Un cliente pide una operación que requiere la colaboración de diversos objetos.
- Problema: Determinar qué objetos tienen que colaborar directamente con el cliente.
- Ventajas, inconvenientes y limitaciones: Disminuye el acoplamiento entre el cliente y las clases que implementan el servicio que se pide.

Expert
- Tema: Responsabilidad
- Contexto: Un servicio utiliza una determinada cantidad de información.
- Problema: Determinar qué clase tiene que ser responsable del servicio.

- Ventajas, inconvenientes y limitaciones: Las clases son más coherentes, se ahorran llamadas entre clases y se disminuye el acoplamiento entre las clases.

Decorator
- Tema: Interfaz
- Contexto: Los objetos de una clase pueden tener que hacer muchas operaciones diferentes, muchas de las cuales o bien se hacen raramente, o bien no se conocen hasta la hora de la ejecución.
- Problema: Añadir operaciones y atributos a un objeto en tiempo de ejecución.
- Ventajas, inconvenientes y limitaciones: En la definición de la clase podemos poner las operaciones "regulares", mientras las especiales aparte; si proliferan los objetos decoradores, el diseño pierde claridad; un objeto y sus decoradores no tienen la misma identidad lo cual limita la posibilidad de utilizar técnicas basadas en ésta.

Factory Method
- Tema: Instanciación
- Contexto: Creación de un objeto del cual se desconoce la subclase a que pertenece.
- Problema: Instanciar un objeto de una clase y determinar la subclase en tiempo de ejecución. Esta situación se presenta a menudo en construcciones en las que hay clases abstractas de las cuales se pide que instancien objetos desde dentro de la construcción misma.
- Ventajas, inconvenientes y limitaciones: Es una opción más flexible que la instanciación directa por la superclase.

Forwarder-Receiver
- Tema: Responsabilidad
- Contexto: Estructura de igual a igual.
- Problema: Desacoplar las clases que colaboran entre sí en los mecanismos de comunicación.
- Ventajas, inconvenientes y limitaciones: La presencia del emisor y el receptor hace el proceso un poco menos eficiente a causa de los mensajes adicionales; como las funciones y datos para la comunicación se encapsulan en los emisores y receptores, un par no resulta afectado por cambios en la ubicación física de los otros pares que utiliza; lo malo es que no soporta fácilmente los cambios dinámicos en la distribución de los pares.

High Cohesion
- Tema: Responsabilidad
- Contexto: Distribución de los atributos y operaciones entre diversas clases.
- Problema: Realizar la distribución mencionada de manera que la complejidad del conjunto de clases sea mínima.
- Ventajas, inconvenientes y limitaciones: El diseño es más comprensible, el mantenimiento más sencillo, el acoplamiento entre clases es menor y además las clases son más fáciles de reutilizar.

Implementing Generalizations
- Tema: Especialización
- Contexto: Clasificación múltiple y dinámica.
- Problema: Clasificar los objetos de una clase por diversos criterios al mismo tiempo, sin que se multiplique el número de subclases.

Keyed Mapping
- Tema: Responsabilidad
- Contexto: Clase asociativa
- Problema: Se accede a un objeto de una clase asociativa desde uno de los extremos. Se quiere saber cuál es el objeto del otro extremo de la asociación sin tener que acceder a él.

- Ventajas, inconvenientes y limitaciones: Ahorra accesos al otro extremo de la asociación; por el contrario, el atributo identificador del extremo repetido en la clase asociativa ocupa espacio y se tiene que actualizar.

Low Coupling
- Tema: Responsabilidad
- Contexto: Distribución de los atributos y operaciones entre clases
- Problema: Hacer la distribución de los atributos de manera que las dependencias entre las clases sean mínimas.
- Ventajas, inconvenientes y limitaciones: Las clases se ven menos afectadas por los cambios en las otras; la función de cada clase es más fácil de entender, reutilizar las clases es más fácil.

Master-Slave
- Tema: Cliente/Servidor, Responsabilidad
- Contexto: Distribución de funciones entre subtareas
- Problema: La distribución de las funciones entre las subtareas tiene que ser transparente a los clientes del conjunto de éstas. Puede hacer falta coordinación entre las subtareas y puede ser conveniente que coexistan diversas implementaciones de la misma función.
- Ventajas, inconvenientes y limitaciones: Se pueden sustituir y añadir esclavos de manera transparente al cliente, los esclavos pueden funcionar en paralelo; no siempre es aplicable: si la distribución del trabajo entre los esclavos y su creación y gestión exigen un consumo de recursos que probablemente no compense otras ventajas, sobre todo si no hay ejecución paralela de los esclavos.

Mediator
- Tema: Cliente/Servidor
- Contexto: Un conjunto de objetos con muchas interacciones entre sí
- Problema: Evitar que la implementación de las clases dependa de las clases con las que tiene que interactuar cada uno.
- Ventajas, inconvenientes y limitaciones: Se pueden reutilizar las clases colegas con mediadores diferentes y los mediadores con clases colegas diferentes; se sustituyen relaciones de varios a varios por relaciones de uno a varios.

Memento
- Tema: Responsabilidad
- Contexto: Token
- Problema: Un objeto recuperable.
- Ventajas, inconvenientes y limitaciones: Se guardan los valores de los atributos del objeto recuperable sin acceder directamente a él, es decir, respetando la encapsulación; descarga la clase recuperable de las operaciones para guardar el estado de los objetos propios y por último, puede ser costoso si hay que guardar objetos grandes a menudo.

Multiple Access Levels
- Tema: Interfaz
- Contexto: Clases con operaciones de acceso restringido
- Problema: Tener más niveles de acceso a las operaciones de una clase que los que permiten las opciones estándar de visibilidad.
- Ventajas, inconvenientes y limitaciones: Permite que haya diferentes niveles de acceso público a una clase; si el lenguaje no soporta las interfaces, la programación se complica.

Observer
- Tema: Responsabilidad
- Contexto: Los cambios en un objeto se tienen que reflejan en otros objetos
- Problema: Actualizar automáticamente el estado de unos objetos A, B, etc. cada vez que cambia el estado de un objeto X.

- Ventajas, inconvenientes y limitaciones: No es necesario que el sujeto conozca la clase de los observadores, lo cual proporciona una mayor flexibilidad al diseño. Se pueden añadir y suprimir observadores fácilmente en tiempo de ejecución; cuando se actualiza el sujeto, no es posible predecir la extensión de las repercusiones.

Prototype
- Tema: Instanciación
- Contexto: De una clase se pueden tener que crear muchos objetos con mucho en común.
- Problema: Crear objetos como copias de un prototipo.
- Ventajas, inconvenientes y limitaciones: Como los prototipos se pueden crear en tiempo de ejecución, este patrón ofrece una opción equivalente a la creación dinámica de clases.

Proxy
- Tema: Interfaz
- Contexto: Un objeto es costoso de instanciar o resulta costoso acceder a él una vez instanciado.
- Problema: Tener un objeto creado o no, que sustituya a algún efecto el objeto principal. Esta sustitución tiene que ser ventajosa desde el punto de vista de la eficiencia y transparente a los clientes.
- Ventajas, inconvenientes y limitaciones: El hecho de que no siempre sea necesario instanciar el sujeto real aumenta la eficiencia del software. En la modalidad de objeto remoto, hace transparente al cliente la localización del sujeto real.

Pure Fabrication
- Tema: Responsabilidad
- Contexto: Diversos procesos relacionados y, eventualmente datos relacionados con éstos, que no se pueden asignar a ninguna clase significativa.
- Problema: Asignar a los procesos a una clase adecuada, ya que en orientación al objeto hace falta que todos los datos sean atributos de alguna clase, y todos los procesos, operaciones de alguna clase.
- Ventajas, inconvenientes y limitaciones: Permite resolver la asignación de responsabilidades en determinados casos dudosos; y la clase creada puede ser totalmente artificial, lo cual disminuye la claridad del diseño.

Singleton
- Tema: Instanciación
- Contexto: Una clase que nunca puede tener más de un objeto al mismo tiempo.
- Problema: Garantizar que una clase no pueda tener más de un objeto.
- Ventajas, inconvenientes y limitaciones: Se puede controlar fácilmente el acceso al objeto único; evita tener que recurrir a variables globales para contener los valores de los atributos del objeto; se puede modificar fácilmente para que el número de objetos pueda ser diferente de uno.

State
- Tema: Responsabilidad
- Contexto: Objeto con diversos estados disponibles.
- Problema: Una operación del objeto tiene que ser diferente según el estado de éste y, por lo tanto, ha de cambiar en tiempo de ejecución cuando el objeto cambie de estado.
- Ventajas, inconvenientes y limitaciones: Todas las operaciones de un estado se reúnen en un solo objeto. Las transiciones entre estados se convierten en explícitas.

Strategy
- Tema: Responsabilidad

- Contexto: La misma operación se tiene que implementar con diferentes algoritmos según los casos.
- Problema: Sustituir un algoritmo por otro sin afectar al cliente.
- Ventajas, inconvenientes y limitaciones: Permite la sustitución dinámica de la implementación de una operación; es una solución más clara que tener una sola operación con condiciones dentro; se pueden tener diversas implementaciones de la misma función, cada una de las cuales en preferible en unas circunstancias diferentes; los clientes tienen que ser capaces de escoger la estrategia; aumenta el número de objetos y disminuye la eficacia.

View Handler

- Tema: Responsabilidad
- Contexto: Un sistema de software que tiene que soportar diversas vistas de la información o que tiene que ofrecer la posibilidad de trabajar con varios documentos al mismo tiempo.
- Problema: Soportar funciones para abrir, cerrar y manipular las vistas. Los cambios introducidos sobre una vista se tienen que propagar a las otras. La implementación de cada vista debe ser independiente de la de las otras y de la implementación de la gestión de las vistas.
- Ventajas, inconvenientes y limitaciones: La gestión de las vistas es uniforme y se puede realizar a la conveniencia de cada aplicación. Cada vista se puede cambiar sin que resulte afectada ninguna otra clase más que su suministrador, la existencia de éste y el gestor disminuye la eficiencia; en general esta solución solo es ventajosa cuando hay que coordinar muchas vistas diferentes o si bien éstas deben poder adaptarse a diversos suministradores.

Abstract Factory

- Tema: Instanciación
- Contexto: Objetos que se pueden implementar de diferentes maneras.
- Problema: Instanciar objetos sin tener que especificar la forma de implementación de antemano.
- Ventajas, inconvenientes y limitaciones: Se puede modificar la forma de implementación de los objetos sin que el cliente se vea afectado. En cambio, es difícil añadir tipos de objeto nuevos, ya que se tendría que modificar tanto la clase fábrica como todas sus subclases. Garantiza que todos los objetos se implementarán de la misma manera si en u momento determinado sólo hay un objeto fábrica.

Chain of Responsability

- Tema: Responsabilidad
- Contexto: Hay una operación común a diversas clases
- Problema: Determinar automáticamente qué clase tiene que ejecutar la operación.
- Ventajas, inconvenientes y limitaciones: Evita que los clientes tengan que determinar qué clase tiene que realizar la operación. Se puede variar la asignación de responsabilidades a los objetos en tiempo de ejecución. Como no se designa explícitamente el objeto responsable de la operación, podría ser que finalmente no lo ejecutara ninguno de éstos.

Client-Dispatcher-Server

- Tema: Cliente/Servidor
- Contexto: Entorno Cliente/Servidor en el cual hay altas, bajas y cambios de dirección de los servidores.
- Problema: Conseguir que los clientes no resulten afectados por los cambios mencionados.
- Ventajas, inconvenientes y limitaciones: Se pueden cambiar los servidores o añadir algunos, con las actualizaciones correspondientes en el registro del consignatario sin afectar a los clientes. Los clientes tampoco resultan afectados si un servidor cambia de posición dentro de una red de ordenadores; en cambio las modificaciones del consignatario pueden tener mucha repercusión; un

servidor se puede sustituir rápidamente por otro y eso fortalece al sistema; por último reconocer que la intervención del consignatario resta eficiencia al sistema.

Command

- Tema: Interfaz
- Contexto: Se pide una operación sin conocer su firma, es decir, sus eventuales parámetros y el tipo de su eventual valor de retorno.
- Problema: Implementar una llamada a una operación mediante un objeto que se puede construir, manipular e interpretar.
- Ventajas, inconvenientes y limitaciones: No hace falta que el invocador sepa el nombre exacto de la clase y de la operación, si el cliente los conoce (esto aísla el invocador de los cambios en el receptor); con las órdenes se pueden realizar todas las manipulaciones propias de los objetos y añadir órdenes nuevas es muy sencillo.

Command Processor

- Tema: Interfaz
- Contexto: La interfaz de usuario se tiene que poder modificar y ampliar fácilmente.
- Problema: Hacen falta funciones generales relativas a las órdenes, como poder deshacerlas, rehacer, interrumpir, registrar, pedir de diversas maneras, agrupar en macros.
- Ventajas, inconvenientes y limitaciones: Es posible pedir las mimar órdenes de diferentes maneras, por ejemplo, por menú y por teclado. Las órdenes se pueden añadir y modificar sin que, en principio, ello repercuta en los clientes; las órdenes se pueden ejecutar en paralelo.

Composite

- Tema: Agregados
- Contexto: Un objeto compuesto y operaciones que se tienen que ejecutar para todos sus componentes.
- Problema: Representar la estructura en árbol de un objeto compuesto con componentes que también pueden ser objetos compuestos y permitir la ejecución recurrente de operaciones.
- Ventajas, inconvenientes y limitaciones: Hay una interfaz única para todo el objeto compuesto; sin embargo los componentes tienen autonomía (operaciones propias). Evita atributos con valores múltiples y de tipo objeto, que serían necesarios si todo el objeto compuesto se implementara como un objeto único; es difícil restringir la clase de los componentes.

Facade

- Tema: Interfaz
- Contexto: Un conjunto de clases relacionadas.
- Problema: Definir una interfaz de alto nivel única para todo el conjunto de clases.

Flyweight

- Tema: Agregados, instanciación.
- Contexto: Objetos pequeños muy numerosos y repetidos.
- Problema: Evitar que cada uno de los objetos se almacene muchas veces.
- Ventajas, inconvenientes y limitaciones: Se ahorra memoria, porque se evitan las repeticiones del estado intrínseco y probablemente también en la representación de este.

Individual Instance Method

- Tema: Responsabilidad
- Contexto: Una clase con objetos heterogéneos.
- Problema: Una operación se tiene que aplicar sólo a uno de los objetos de una clase.

Iterator

- Tema: Agregados
- Contexto: Un agregado de objetos (una lista, un árbol, etc).
- Problema: Recurrir un agregado de objetos de diversas maneras sin acceder al estado y sin tener que incorporar las operaciones de recorrido al agregado mismo.
- Ventajas, inconvenientes y limitaciones: El mismo agregado se puede recurrir en diversos órdenes, por ejemplo, un árbol se podría recurrir en preorden postorden, etc. La interfaz del agregado es más simple, porque no tiene que incluir operaciones como primero… Nada impide que haya diversos objetos de iterador 1 al mismo tiempo, lo cual significa que puede haber diversos recorridos independientes del agregado en cuarso al mismo tiempo.

Template Method

- Tema: Interfaz
- Contexto: Un algoritmo ligado a una clase ha de tener algunas variaciones en el nivel de las subclases.
- Problema: La parte común del algoritmo se implementa en el nivel de clase y las partes dependientes de la subclase están implementadas en éstas.
- Ventajas, inconvenientes y limitaciones: Es una de las técnicas más empleadas para reutilizar código.

Visitor

- Tema: Agregados
- Contexto: Un objeto compuesto
- Problema: Añadir y modificar operaciones que se tienen que hacer sucesivamente con todos los elementos del agregado, sin verse obligado a modificar las clases de estos elementos.
- Ventajas, inconvenientes y limitaciones: Hace que las clases de elementos sean independientes de los tratamientos sobre éstos (favorece la reusabilidad); añadir nuevas operaciones sobre los elementos es muy fácil ya que basta con añadir una subclase nueva a visitor; en cambio añadir clases de elementos nuevas es difícil porque puede exigir modificar todas las clases de visitadores; los visitadores pueden acumular fácilmente información obtenida durante el recorrido de los elementos del agregado.

Interpreter

- Tema: Agregados
- Contexto: Un lenguaje con una gramática sencilla.
- Problema: Implementar el procesamiento de expresiones en un lenguaje orientado al objeto.
- Ventajas, inconvenientes y limitaciones: Sólo es aplicable a lenguajes de gramática sencilla, de lo contrario haría falta un gran número de clases; probablemente no será la técnica más adecuada cuando la eficiencia tiene mucha importancia.

TEMA 3 DESARROLLO DE INTERFACES GRÁFICAS DE USUARIO

Tema 3
Desarrollo de interfaces gráficas de usuario

1. Concepto de desarrollo de interfaces gráficas de usuario
2. El factor humano
3. Teorías y principios
4. El análisis de las tareas
5. Análisis y diseño de las interfaces gráficas

1. CONCEPTO DE DESARROLLO DE INTERFACES GRÁFICAS DE USUARIO

La existencia y el funcionamiento de los ordenadores y de todo el software, sólo se justifica en la medida en que toma datos del mundo exterior y a su vez lo vuelve e enviar a éste tras un procedimiento o cálculo; y en la mayoría de las ocasiones este envío es a la pantalla del ordenador. La parte que capta la información y la forma en que posteriormente es enviada una vez procesada es la interfaz del sistema informático.

A la hora de construir la interfaz gráfica, ésta debe ser sólida, rápida y fiable, pero además debe tener en cuenta aspectos psicológicos y ergonómicos para ayudar a los usuarios en sus actividades de trabajo u ocio.

Se pretende que la interfaz de usuario sea fácil de aprender y recordar, sea productiva en el trabajo, evite cometer errores y dé satisfacción a los usuarios; aunque según el software existen prioridades distintas en estos objetivos. Así en un sistema crítico para la seguridad es menos importante la rapidez en el aprendizaje y la satisfacción en el usuario y es fundamental que evite cometer errores; casi lo contrario puede ocurrir en el software comercial y juegos, donde el aprendizaje debe ser rápido, y debe ser satisfactorio para los usuarios.

La **interfaz gráfica de usuario** es aquella interfaz que permite al usuario navegar por la información de su ordenador e interactuar apuntando, cliqueando y arrastrando iconos y otros datos dentro de la pantalla por medio de un ratón, en lugar de utilizando palabras y frases.

2. EL FACTOR HUMANO

Estudiaremos aquí los factores psicológicos, ergonómicos y sociales de la interfaz de usuario:

- **Percepción y representación**: La percepción consiste en interpretar la información que nos llega del exterior por los órganos de los sentidos y relacionarla con la experiencia. La percepción en informática es eminentemente visual y existen dos teorías sobre ella: las teorías constructivistas dicen que no vemos una copia objetiva de la realidad, sino que el sistema visual transforma y distorsiona la información a partir de nuestra experiencia anterior; las teorías ecológicas, en cambio, consideran que la información no se construye, sino que directamente se detecta.
- **Atención**: Consiste en percibir una parte concreta de la información que recibimos por los sentidos. Hay 3 tipos de atención: la **sostenida o vigilancia**, que trata de detectar los cambios que se producen más que analizarlos; la **concentrada** que consiste en prestar atención a un único acontecimiento de entre todos los posibles; y la **atención dividida** que es lo que sucede, por ejemplo, cuando se conduce y se habla al mismo tiempo. Con respecto a la interfaz de usuario decir que la información más urgente se tiene que colocar en lugar preeminente para que sea rápidamente captada por el usuario, mientras que la menos urgente se sitúa siempre en el mismo lugar para que el usuario sepa dónde encontrarla cuando la necesite; la información que se necesita ocasionalmente no debe presentarse permanentemente, sino solo cuando se solicite.
- **Memoria**: Hay 3 tipos de memoria, la **memoria sensorial** recibe la información que nos llega directamente de los órganos de los sentidos, se percibe un gran volumen de información y por ello en la pantalla sólo debe aparecer lo necesario para no fatigar al usuario; la **memoria a corto plazo** contiene la información reconocida y percibida, así como la transferida desde la memoria a largo plazo, es la memoria de trabajo y dura unos 10-20 segundos y tiene una capacidad bien conocida, de 5 a 9 elementos; la **memoria a largo plazo** contiene todo lo que potencialmente somos capaces de recordar. Hay muchos factores que afectan a la memoria y al recuerdo, y los símbolos

gráficos son más fácilmente recordables que el texto, siempre que haya un buen enlace entre el símbolo y el objeto, proceso o entidad que representa.

- **Recordar y reconocer**: Es mucho más fácil reconocer una cosa que recordarla; por lo que es mucho más fácil pedir una función por medio de un menú que desde una línea de comando.
- **Tiempo de respuesta**: Es el tiempo que el ordenador tarda en presentar el resultado de una acción del usuario. Para determinadas tareas como el clic de ratón, el desplazamiento por la pantalla, etc, debe ser muy bajo, de lo contrario el usuario se fatigará; pero en contra de lo que parece no siempre el tiempo de respuesta óptimo es el más bajo; así si un determinado proceso tarda un 75% menos en producirse de lo habitual, creará ansiedad en el mismo, dado que pensará que la orden no se ha ejecutado correctamente.
- **Conocimiento y Modelos mentales**: El conocimiento está muy bien organizado en nuestra mente, basta con ver la facilidad con que podemos responder instantáneamente muchas preguntas. Los **modelos mentales** se construyen dinámicamente a partir de los esquemas mediante la experiencia y el entrenamiento. Nos establecemos modelos mentales para todos los objetos que nos rodean, principalmente dos tipos de modelos: estructurales en los que se describe el funcionamiento de algo en términos de sus componentes; y funcionales que describen el comportamiento de un dispositivo en términos de funciones conocidas.
- **Metáforas y Modelos conceptuales**: El razonamiento metafórico consiste en utilizar conocimiento anterior para entender situaciones nuevas. Las **metáforas verbales** tienen una función introductoria, es por ejemplo utilizar un procesador de textos como si de una máquina de escribir se tratase. Las **metáforas virtuales** son parte de la interfaz y constituyen el modelo que se aprende, así la papelera de reciclaje tiene la misma función e icono que en la vida diaria.
- **Tipos de errores**: Hay muchos tipos de errores, de concepto (no entender correctamente el funcionamiento), de captura (utilizar por descuido una función que se utiliza muy a menudo en detrimento de la que realmente queríamos utilizar), de descripción (cerrar una ventana cuando realmente queríamos minimizarla), de pérdida de activación (tras una interrupción olvidamos qué es lo que estábamos haciendo), de modo (cuando tecleamos en el procesador de textos creyendo que insertamos caracteres nuevos y en realidad estamos sobrescribiendo los que ya existen).
- **Aprendizaje**: Es el proceso mediante el cual el usuario se crea un modelo conceptual del sistema del software. El problema de comenzar a utilizar el software es que inicialmente hay que leerse el manual de instrucciones, cosa que nadie hace; hay que buscar ayuda contextual que muchas veces nos confunde más que ayudarnos, a veces el usuario "interpreta" lo que el software debe hacer o directamente se lo inventa.
- **Diversidad de los usuarios**: Dependiendo del tipo de personalidad, edad, sexo, tipo de trabajo a desarrollar con el software, conocimientos informáticos anteriores, etc. hay una gran diversidad de usuarios y para todos ellos está diseñado el mismo software (comercial), de ahí el problema.

3. TEORÍAS Y PRINCIPIOS

Equilibrio entre automatización y control para los usuarios

- Evitar los cambios de modo frecuentes y hacer el mínimo uso de las ventanas modales que obligan al usuario a responder antes de poder continuar su tarea.
- Los asistentes tienen que guiar, pero no forzar al usuario. En se mismo sentido conviene permitir interrupciones en el máximo número de tareas posibles y que el usuario pueda reanudarla o detenerla.
- Retroalimentación al usuario de las acciones realizadas y, si es posible, poder deshacerlas.

- Hay que diseñar la interfaz para distintos niveles de pericia, y además que en la medida de lo posible, sea adaptable a sus preferencias.
- Incluso en tareas largas de secuencias automatizadas, conviene informar al usuario del progreso de la acción en curso para darle al menos la impresión de un cierto control y permitirle la cancelación en cualquier momento.

Minimización de la carga de memoria del usuario

- No conviene hacer recordar al usuario aquello que el ordenador puede recordar por él.
- Reconocer mejor que recordar, por ejemplo presentar menús con listas de valores.
- Poner recordatorios de la aplicación, documento, modo y parámetros como el tipo de letra; los botones seleccionados conviene que se muestren hundidos.
- Permitir el uso de códigos mnemónicos, especialmente cuando son un están dar, como CTRL+P para imprimir.
- Utilizar metáforas que simulen el mundo real, como la papelera de reciclaje.
- Aplicar los principios de diseño como agrupación para obtener claridad visual.
- Diseñar el interfaz según el modelo objeto acción: se selecciona un objeto y después se despliega el menú de acciones permitidas.

Consistencia

- La consistencia de la interfaz de usuario contribuye de manera decisiva a lograr que el software sea mucho más fácil de aprender y que el riesgo de errores disminuya considerablemente.
- Consistencia en el contexto de tareas: títulos de ventana, árboles para la navegación, ayudas y menú contextuales.
- Consistencia entre diferentes productos de software en cuanto a presentación, comportamiento de la interfaz, técnicas de interacción…
- Consistencia entre las versiones sucesivas de un mismo producto.
- Consistencia con respecto al uso de colores, tipo de letra e iconos.

Pautas para la presentación de la información

- Presentación coherente con respecto a abreviaciones, formatos, colores, uso de mayúsculas…
- Agrupación y disposición en columnas de los datos, alineación adecuada de los valores, especificación de unidades de medida.
- El formato en que se presenta la información tiene que ser lo más parecido al formato en que se introduce.
- Utilizar solo dos niveles de brillo, no más de tres tipos de letra ni más de cuatro tamaños.
- El parpadeo y la intensidad de color se tiene que utilizar de manera limitada y en áreas reducidas; cualquier técnica destinada a destacar, si se usa en exceso, hará que nada destaque.

Pautas para la entrada de datos

- Hacer un uso consistente de los delimitadores, abreviaciones, etc.
- Minimizar el tecleo y el movimiento del ratón, es decir, seleccionar listas en vez de teclear, proponer valores y, especialmente, no obligar al usuario a introducir dos veces la misma información, aunque se le debe permitir cambiarla si es lo que quiere.
- Dar una cierta flexibilidad en el orden de entrada de datos, aunque esta práctica va en contra de la consistencia.

4. EL ANÁLISIS DE LAS TAREAS

El análisis de las tareas trata de describir las tareas que los usuarios harán con el software futuro. Una tarea se puede definir como lo que hace una persona con vistas a conseguir una meta mediante algún dispositivo. Las tareas se pueden descomponer en subtareas de diversos niveles y éstas, finalmente en acciones. Por último, una meta es un estado del sistema que una persona quiere conseguir.

Para analizar las tareas, nos encontramos con distintos aspectos que debemos cubrir:

- **Análisis de los flujos de trabajo**: Consiste en descomponer una tarea compleja en la que participan diversos usuarios, en un esquema de subtareas alternativas, en secuencia y condicionales, es decir un flujo de trabajo o workflow. Para describir estos flujos de trabajo podemos utilizar los diagramas de actividades de UML.
- **Análisis de los trabajos**: Trata de determinar qué hace un usuario concreto durante el día o durante la semana. Nos interesa no solo la lista de tareas, sino también la frecuencia, duración, dificultad de las mismas…
- **Estudio individual de las tareas**: Consiste en estudiar los pasos y decisiones dentro de una tarea hasta llegar al nivel de las acciones.
- **Entorno de los usuarios**: El trabajo de los usuarios se ve afectado por la actividad que hay a su alrededor. Distintos parámetros determinan este entorno: la electricidad disponible, si hay lugar para teclado y ratón y para abrir manuales, la iluminación, problemas de suciedad o polvo, si cada usuario tiene su equipo de trabajo individual o es compartido, acceso a Internet, jerarquía de los usuarios, entorno cultural, extracto social y demográfico, cultura del país de referencia, nivel salarial del usuario…

Para analizar todo lo anterior se pueden y deben realizar **visitas a los usuarios** entendiendo que tienen que ser visitas a los usuarios reales de la aplicación a desarrollar y no a sus superiores que tenderán a darnos una visión distinta de la realidad. La recogida de información puede ser por varios sistemas: Observar a los usuarios mientras trabajan preguntándoles todo aquello en lo que dudamos, observarlos y hacerles hablar en voz alta lo que van haciendo en cada momento, grabarlos en audio y vídeo (aunque esto dificulta el análisis posterior pues requiere mucho tiempo para su tratamiento), recoger muestra de impresos y listados del trabajo del usuario, técnicas de reuniones de grupo para conocer las opiniones y mejoras del nuevo software que vamos a crear, entrevistas individuales con cada usuario si no son demasiados, etc.

Tras las visitas hay que analizar esta información, que, como ya dijimos anteriormente, vamos a realizar principalmente por medio de diagramas de flujo de trabajo (diagramas de actividades de UML).

Finalmente debemos cumplir requisitos de funcionalidad del software y requisitos de usabilidad. Éstos últimos consisten en objetivos concretos para satisfacción del usuario. Es decir, plantearse en términos temporales cuánto deben tardar los usuarios en hacer determinadas tareas y en manejarlas completamente. Por ejemplo, con el nuevo software de nóminas, un empleado debe tardar unos 15 minutos en dar de alta a otro trabajador y establecer los salarios y datos del mismo.

5. ANÁLISIS Y DISEÑO DE LAS INTERFACES GRÁFICAS

El objetivo del análisis y el diseño de la interfaz de usuario es llegar a una descripción detallada de la interfaz gráfica en cuanto a los elementos que figuren y en cuanto a la interacción, que se pueda programar en la herramienta escogida.

El **modelo conceptual** de la interfaz de usuario es una lista de los objetos con los cuales trabaja el usuario y las acciones que hace sobre cada uno. Es la base para establecer a grandes rasgos la metáfora del sistema.

A continuación se exponen una serie de normas generales para la elaboración del diseño de la interfaz de usuario:

- **Principio del agrupamiento**: Hay que organizar el espacio en bloques separados con controles similares y con un título para cada uno. Esto lo vemos por ejemplo en Windows, donde órdenes similares se encuentran agrupadas bajo un mismo menú; así el usuario encuentra rápidamente la información que necesita y además le ayuda a formarse un modelo conceptual del modelo del programa.
- **Principio de visibilidad y utilidad**: Los controles utilizados con frecuencia deberán de ser visibles y de fácil acceso y, por tanto, deberán ocultarse o comprimir los menos utilizados. Esto sucede en el menú de los programas de office donde solo aparecen inicialmente los que más utilizamos, ocultándose el resto o con las barras de herramientas en los que aparecen en forma de iconos los más usados.
- **Principio de la consistencia inteligente**: Hay que utilizar una distribución de la información similar para funciones similares. El usuario cuando ha encontrado una información o un control que buscaba, espera en situaciones parecidas hallarlo todo en el mismo lugar. Por ejemplo, cuando navegamos por una web, siempre esperamos encontrar los botones de navegación en el mismo sitio y orden, independientemente de la página en la que nos encontremos.
- **Principio de economía del diseño**: Hay que omitir cualquier elemento que no aporte ninguna información. La pantalla de un ordenador es un espacio limitado y no debemos "llenarla" con información innecesaria o con elementos decorativos que no aporten ni información ni identidad. Buscadores como yahoo y google deben buena parte de su éxito a seguir este principio.
- **Principio del color como suplemento**: El color debe ser usado como medida para enfatizar la información, pero no solamente para comunicarla. UN buen criterio para diseñar una interfaz es diseñarla en blanco y negro y después agregar color donde sea estrictamente necesario. Una buena política es limitarse a los mismos colores en distintos matices para la interfaz y reservar los más brillantes para logotipos, símbolos de identidad o mensajes especiales que requieran nuestra atención. En los laboratorios de Apple se produjeron grandes discusiones sobre si añadir o no color a la interfaz de MacOS.
- **Principio de reducción del desorden**: Es un resumen de los principios anteriores. Si sólo los controles utilizados más frecuentemente están visibles, agrupados con sentido, con un uso minimalista del color y sin elementos superfluos, entonces la interfaz será clara, atractiva y funcional.

Visualización de la información

Tenemos muchos recursos gráficos para utilizar en nuestra interfaz, destacamos los siguientes:

- Alfanuméricos: Podemos utilizarlos tanto como queramos, y es que toda interfaz y toda página web está llena de caracteres alfanuméricos.
- Formas: Si la forma corresponde en el objeto o la operación representada son muy efectivas. Utilizar más de 20 formas puede generar confusión.
- Color: Ya hablamos del principio del color como suplemento. Un uso abusivo provoca fatiga y confusión y se conoce como polución del color. Como mucho debemos utilizar solo 10 colores distintos.
- Ángulo de las líneas: Adecuado para indicar direcciones, aunque teniendo en cuenta que un usuario podrá distinguir en una interfaz unas 8 direcciones.
- Longitud de la línea: Para indicar magnitudes, y solo debemos usar 4-5 distintas.

- Grosor de la línea: Apropiado también para designar magnitudes, e igualmente solo utilizaremos 4-5 distintas.
- Estilo de la línea: Manifiesta diferencias entre los valores que representa.
- Tamaño del objeto: Representará características cuantitativas del objeto representado.
- Intermitencia: Es una indicación visual muy fuerte. No debemos usar más de dos intermitencias en la misma interfaz a la vez y además no deben durar todo el rato, deben tener un final pues cansan mucho.
- Vídeo inverso: Para destacar colecciones de datos es muy útil.
- Subrayado: Al igual que el video inverso destaca datos pero debe utilizarse con moderación, ya que si no fragmentamos mucho la información.

Elementos de diseño

1. Tipografía

Elementos de Diseño

- Tipografía
- Cantidades
- Color
- Iconos
- Imágenes
- Animación
- Organización espacial
- Representación tridimensional

La tipografía es la apariencia que tiene el texto y es el elemento de diseño gráfico más básico que existe; ya se hallaba presente en los monitores de fósforo verde de 24 filas x 80 columnas. En la tipografía destacamos: la fuente que es el tipo de letra que utilizamos, el cuerpo que es el tamaño de la fuente y se mide en puntos o picas (1 pica = 12 puntos), las serifas son una herencia romana y consiste en una pequeña proyección final de los palos de las letras, el peso hace referencia al grosor de la letra (normal o negrita) y la inclinación puede ser vertical o cursiva (también llamada itálica).

Hay varios factores que afectan a la legibilidad del texto:

- **Fuentes proporcionales y de ancho fijo**: Una fuente de ancho fijo es aquella en la que todos los caracteres ocupan la misma anchura, una i lo mismo que una m. Es ideal para columnas de datos, en otro caso utilizamos habitualmente las fuentes proporcionales.
- **Tamaño de la fuente**: Un texto de 10-12 puntos es apropiado, habitualmente en una interfaz con establecer cuatro tamaños distintos de fuentes es suficiente para no saturar.
- **Mayúsculas/Minúsculas**: Un texto en mayúsculas se lee un 12% más lento, dado que encuadra toda palabra en un rectángulo, distinto a las mayúsculas y minúsculas que le dan forma.
- **Espaciado e interlineado**: Habitualmente no cambiaremos el espaciado entre caracteres ni el interlineado.
- **Longitud de la línea**: Una línea muy larga puede obligarnos a mover la cabeza para descubrirla en toda su extensión, si es muy corta fragmenta en exceso el proceso de lectura. Típicamente los textos impresos en un libro tienen una longitud de 60 caracteres y las de una columna de periódico 30.
- **Justificación**: La justificación a derecha e izquierda es más estética pero ralentiza la lectura.
- **Entorno global**: Para la mayor parte de las interfaces tendremos bastante con dos tipos de letra, dos inclinaciones, dos pesos y cuatro tamaños.
- **Maquetación**: Es la distribución de los distintos párrafos de texto y otros elementos gráficos en un espacio determinado; es la primera impresión que nos llevamos de una web y como todas las primeras impresiones es importante.

2. Cantidades

Hay elementos gráficos que permiten expresar las cantidades y sus tendencias mucho mejor que los caracteres numéricos. Así encontramos una ayuda inestimable en los mapas de puntos, gráficos de líneas, gráficos de barras, diagramas circulares.

3. Color

Ya hemos hablado antes del color, pero ahora nos vamos a centrar más en los modelos que existen. Por un lado está el modelo más primitivo el **RYB** que tenía 3 colores primarios (rojo, amarillo y azul) y de la combinación de ellos se generaban el resto, aunque el negro se hacía imposible de conseguir. Otro modelo es el **CMY** se basa en 3 colores primarios también pero son el cyan, magenta y amarillo, con el cual ya se puede obtener perfectamente el negro.

El **CMYK** se basa en los 3 anteriores, pero se utiliza en impresión a tinta, y para imprimir perfectamente el negro, éste se pone aparte (black) en lo que se llama impresión a cuatro tintas o cuadricromía (impresoras de chorro de tinta).

El **RGB** se basa en la refracción de la luz blanca sobre un prisma.

Los colores según su **tonalidad** pueden ser cálidos o fríos. Cuanto más rojo contenga un color, más cálido es. Los colores fríos (más azules) dan la impresión siempre de encontrarse más alejados en el horizonte.

Por último unas breves normas sobre la utilización del color, aunque es conveniente de vez en cuando transgredir alguna de ellas para ver el efecto que se produce en la interfaz, dado que no son normas cerradas e infranqueables:

- Hay que evitar la utilización de colores complementarios muy saturados, evitar el azul puro para el texto (no se distingue bien), evitar el uso de una gama de colores basados en una única tonalidad pues los usuarios con problemas de visión no podrán distinguirlos correctamente.
- Hay que tener en cuenta la diferencia entre el color que se da en pantalla y el impreso.
- Los colores similares expresan significados similares: podemos mostrar relaciones entre elementos asignándoles tonalidades similares.
- Los colores luminosos y saturados atraen la atención del usuario.
- Debemos usar los colores cálidos para acciones que requieran inmediatamente nuestra atención, mientras los colores fríos podremos usarlos para información de estado o en segundo plano.

4. Iconos

Un icono es una representación visual compacta y universal de objetos, funcionalidades y procesos del ordenador. Así podemos representar carpetas, memoria, procesadores, etc. en pequeñas imágenes no mayores de 32x32 píxeles y que pueden ser reconocidas fácilmente por cualquier persona de cualquier cultura. Así un buen icono ahorrará un espacio en pantalla precioso, será reconocido y recordado fácil y rápidamente y ayudará a la internacionalización de la interfaz.

Los iconos pueden representar objetos mediante objetos concretos (igual al que representan, carpetas, reloj) mediante símbolos abstractos (flechas, puntos) o mediante una mezcla de ellos. Si nos fijamos en el mapeo entre lo que representa (el objeto o proceso) y su representación (el icono) tenemos varios tipos de representaciones:

- Iconos de **semejanza**: Muestran el concepto mediante una representación análoga, como por ejemplo la papelera de reciclaje.
- Iconos **ejemplares**: Se muestra el concepto mediante un ejemplo. Un símbolo de tenedor y cuchillo no significa que vendan esos cubiertos, sabemos que quiere decir que es un restaurante.
- Iconos **simbólicos**: Es un nivel de abstracción superior. Por ejemplo para designar objeto frágil se indica con una copa rota.
- Iconos **arbitrarios**: Deben ser aprendidos porque no hay una relación directa, por ejemplo la e de Internet Explorer.

Principios para el diseño de iconos:

- **Consistencia**: Un icono nunca se diseñará aislado de los demás. Los colores, formas, grosores de línea, etc, deben estar en consonancia.
- **Equilibrio**: Hay que diseñar una barra de iconos equilibrada, sin que por saturación, brillo o tamaño destacan unos por encima de otros, sobre todo si no tienen más importancia que los demás.
- **Legibilidad**: La utilización de representaciones grandes, líneas gruesas y áreas simples facilita la legibilidad.
- **Reconocimiento**: Hay que intentar utilizar una metáfora fácilmente reconocida por los usuarios.
- **Economía del color**: Hay que diseñar los iconos primero en blanco y negro y después añadirles como mucho, 3 colores.
- **Condiciones culturales**: Evitaremos los símbolos que puedan tener connotaciones distintas en los diferentes países, pues así luego nos ayudará a la internacionalización de la interfaz. Así un icono con una cruz roja para la ayuda no servirá en países musulmanes donde la ayuda la presta la media luna roja.

5. Imágenes

Las imágenes nos informan, nos atraen, nos interesan y nos comunican. El tamaño de una imagen en pantalla depende de los píxeles que utilice. Como la imagen es discreta (no continua como la fotografía analógica) podemos hacer zoom hasta el nivel de detalle que nos permita sus píxeles. El tamaño de una imagen (peso) dependerá del número de píxeles que lo formen y de la profundidad del color, que puede ir desde los 8 bits (256 colores) hasta los 32 (muchos millones de colores).

6. Animación

Una animación es la sucesión de imágenes en el tiempo, de hecho la simplicidad de esta definición no hace justicia a la complejidad que hay detrás de la misma. La animación es el medio que puede expresar mejor la naturaleza dinámica de determinados objetos y procesos, además cualquier objeto en movimiento captará nuestra atención mucho más rápidamente que uno estático, por eso mismo debemos tener cuidado en su utilización y no "animar" objetos que puedan distraer la atención de lo verdaderamente importante. Los tipos de animación que podemos encontrar son:

- Detalles animados: para atraer la atención del usuario o por motivos estéticos.
- Efectos transicionales: acción o proceso del usuario, como copiar una carpeta en Windows.
- Secuencias cortas: Secuencias de pequeña duración que explican procesos como el funcionamiento de un motor o el crecimiento de una planta.
- Secuencias largas: sobre todo en juegos, donde las secuencias posteriores dependen de la evolución del jugador.

7. Organización espacial

Todo lo que el diseñador ponga en el diseño de una interfaz debe estar justificado, tener una consistencia y seguir una organización espacial, diseñar va en contra de lo casual y desorganizado. Todos los elementos gráficos de nuestra interfaz están dispuestos sobre una retícula imaginaria que se respeta siempre a lo largo de la aplicación, el tamaño y número de divisiones de la retícula es crucial y sigue el recurrente número mágico de Millar 7+/-2 que es el número máximo de elementos que el ser humano puede retener en la memoria a corto plazo.

8. Representación tridimensional

Las representaciones 3D en la **interfaz** están a la orden del día, muchos botones parecen estar en relieve para que inviten a ser apretados y muchas barras permiten ser desplazadas. La luz parece provenir de arriba que es la zona natural de donde proviene la luz en la atmósfera. Al utilizar una interfaz, el conocimiento previo que tenemos de un mundo tridimensional es crucial. Las técnicas para conseguir esta representación 3D en pantallas bidimensionales son:

- Tamaño: Cuanto mayor sea un objeto igual a otro, más cercano nos parecerá.
- Interposición: Si un objeto esconde parcialmente a otro, el segundo se percibe como si estuviera detrás.
- Contraste y claridad: los objetos borrosos y poco contrastados se perciben más alejados.
- Color y Textura: Los objetos en color azul grisáceo tienden a percibirse más alejados, al igual que los objetos que pierden textura.

Cuando lo que es en 3D es **toda la interfaz** solemos hablar de juegos tridimensionales como el mítico Doom. Para que estos juegos sean creíbles, debe tenerse en cuenta:

- Paralaxis de movimiento: Cuando nos desplazamos los objetos más cercanos se mueven a más velocidad que los más alejados, pensemos sino en lo que vemos en una ventana de un tren en marcha.
- Nivel de detalle: Los elementos más alejados se ven más borrosos, menos contrastados y con menos nivel de detalle que los más cercanos.
- Calidad de la representación: Hay que llegar a un acuerdo entre la calidad de lo que queremos representar y el tiempo de cálculo. El ordenador tarda mucho en "dibujar" las representaciones 3D, por tanto si dibujamos demasiado nivel de detalle, se perderá la sensación de realismo por la lentitud de estas representaciones.
- Fotogramas por segundo: La cantidad mínima de fps para dar sensación de continuidad al ojo humano es de 15, 20 es aceptable, en el cine se utilizan 24 y en el vídeo 25.
- Tiempo de latencia: Es el tiempo que transcurre entre que comunicamos una orden al ordenador y percibimos el resultado. Si este tiempo es muy elevado, no tendremos la sensación de que el ordenador responde a órdenes nuestras y con la pérdida de fluidez habrá pérdida de realismo.

Texto elaborado a partir de:
Ingeniería del software orientado al objeto
Benet Campderrich Falgueras, Rosa Valcárcel Larriba
Junio 2005