5 de junio de 2017

Ingeniería del Software I

Samuel García
Ignacio Ballesteros

Índice general

Bloque 1

Introducción a la Ingenería del Software

Ingenería del Software (*IEEE*) La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de *software*.

Las **Ciencias de la Computación** se preocupa de los fundamentos de la teoría; mientras que la Ingeniería del Software de los aspectos prácticos.

La **Ingeniería del Software** estudia los **productos** producidos (ejecutables, módulos, sistemas, liberías...) y los **procesos** usados para producir esos productos.

1.1. La crisis del Software

Los principales problemas que hay detrás de la crisis del Software son:

- El incremento en el tamaño y la complejidad
- Los sobrecostes
- Fallos en el diseño
- Mal mantenimiento
- Herramientas no solo de programación.

Pero desde un enfoque más moderno, tambien se aprecian otros tipos de problemas:

- Falta de robustez en el Software para componentes críticos o de los que somos dependientes.
- Excesiva complejidas en el Software como para entenderlo y comprenderlo.
- Exigencia de cambiar rápidamente.

Frente a esto, desde la Ingeniería del Software se plantea la mejora en las metodologías y el uso de lenguajes de alto nivel (mayor abstacción).

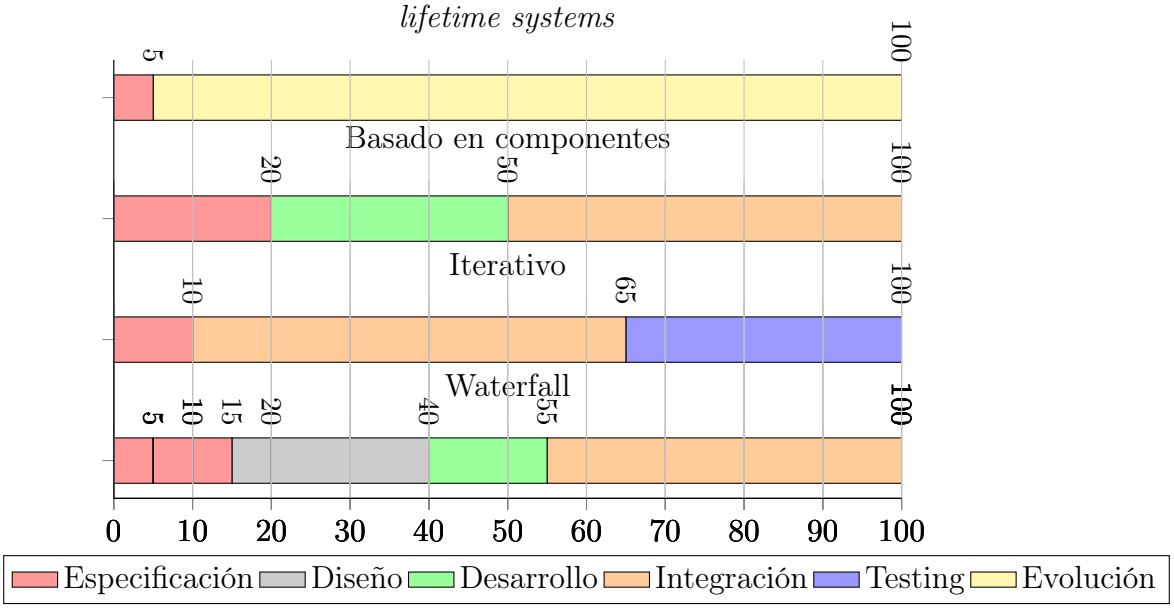
Tras el intento de solucinar esto problemas, el concepto de Ingeniería del Software se redefine en:

Disciplina que tiene como objetivo la producción de Software libre de errores, sin retrasos y dentro del presupuesto; satisfaciendo las necesidades del cliente

1.2. Costes en la Ingeniería del Software

En cuanto a la realización del productos, aproximadamente el 60 % del coste es el desarrollo, y el 50 % en pruebas (test). Sin embargo, con el uso del software, los gastos de mantenimiento superan a los del desarrollo.

Figura 1.1: Distribución del coste según metodología *lifetime systems*



Cuanto más tarde se encuentran errores en el Software, o más tarde se hace un cambio de requisito, los cambios son más costosos. Comparando el cambio en dos fases del ciclo de vida:

Temprano	**Tardío**
Cambio en la especificación.	Cambio en la especificación. Cambiar el código y la documentación. Probar el cambio. Testing. Instalación del producto en el cliente.

A medida que pasa el tiempo, la cantidad de fallos que aparecen aumentan debido al deterioro.

1.3. El Software

Entre las características que se le atribuyen a un *producto Software* encontramos:

- Múltiples **programas**.
- Archivos de **configuracion**.
- La **documentación** del sistema.
- La documentación de **uso**.
- Los **datos** del sistema.
- **Actualización** de información.

El *Software* se realiza para **clientes particulares** o para el uso **general**. Esto está también relacionado con que los productos de software sean **genéricos** o **a medida**.

Dependiendo del producto desarrollado, el software puede entrar en categorías como: tiempo real, negocios, científico, emebido, PC, IA, Web...

Los atributos de un **buen Software** varían según las perspectivas:

Usuario	**Desarrolador**
Exactitud	Consistencia
Confiabilidad	Comprensibilidad
Eficiencia	Capacidad de ser probado
Mantenibilidad	Compacidad
Usabilidad	Compatibilidad
Robustez	

1.4. Procesos en el Software

Se entiende por *proceso Software* un conjunto de **actividades y resultados** asociados a la producción de Software.

Este proceso se puede analizar desde diferentes perspectivas:

- Flujo de **trabajo**.
- Flujo de **datos**.
- **Acción**.

Los modelos del *ciclo de vida* del Software especifican las fases del *proceso de Software*. Hemos mencionado ya ejemplos en la sección 1.2. Un modelo está compuesto de:

- Descripción propia.
- Reglas.
- Recomendaciones (*guías de estilo*).
- Procesos (*actividades a seguir*).

Estos **modelos** están orientados a resolver los retos de la ingeniería del Software, muy relacionados con los atributos del buen Software indicados en la sección 1.3.

- Heterogeneidad de plataformas.
- Entrega más rápida.
- Confianza.
- Gastos en el Hardware/Software.
- Adaptabilidad a nuevas tecnologías.
- Usabilidad.
- Mantenimiento.

1.5. Ciclo de Vida

Qué hacer →Cómo hacerlo →Hacerlo →Probarlo →Usarlo →Mantenerlo

Para la realización del Software tendremos que tener en cuenta:

- Escala
- Productividad
- Calidad (ISO)
    - Funcionalidad, Fiabilidad, Usabilidad, Eficiencia, Manteniblidad, Portabilidad.
- Consistencia
- Tasa de cambio

Este ciclo de vida se organiza en fases. A cada fase se obtendrá un resultado que se utilizará en las siguientes fases. El *ciclo de vida* del Software se enfoca en manejar la complejidad y el cambio a lo largo de un largo proceso.
Existen distintos *ciclos de vida*, nombrados como modelos (sección 1.3):

- Informal
- Convencional
- Incremental
- Evolutivo
- Prototipado (puede ser incluido en los anteriores modelos)

Estos modelos tienen un equilibrio entre los siguietnes factores:

- Velocidad de desarrollo
- Calidad
- Visibilidad
- Sobrecarga de gestión
- Exposición al riesgo
- Relaciones públicas

Bloque 2

Ingenería de Requisitos

Bloque 3

Diseño Estructurado de Alto Nivel

Bloque 4

Objetos

4.1. Orientación a objetos

4.1.1. Enfoque estructurado

Se denomina enfoque estructurado a la forma de pensar el software en términos de funciones de transformación de datos (se disocia entre funciones y datos, y las tareas se interpretan como una transformación de los últimos).

Ejemplo: Pintar un círculo

El enfoque estructurado resuelve el problema de pintar un círculo de la siguiente forma:

- Usa una definición de círculo que esté acorde con los recursos de software (en este caso la expresión algebraica).

$$R^2 \leq (x - x_0)^2 + (y - y_0)^2 \tag{4.1}$$

  donde el radio R y las coordenadas del centro son las constantes que especifican un círculo concreto.

- Disocia la definición de círculo en dos partes y las reinterpreta:
  - Considera que R y el centro son datos para pintar el círculo y añade el color.
  - Convierte la expresión declarativa en una función operativa que transforma el conjunto de datos precedentes en (x, y, color) de todos los píxeles para pintar el círculo en la pantalla.

- Como resultado final se obtiene un sistema capaz de pintar un círculo en términos de un proceso de transformación de datos.

El sistema software se expresa como una función F(x) que transforma el conjunto de datos (R, x_0, y_0) en otro conjunto de datos, en este caso de píxeles.

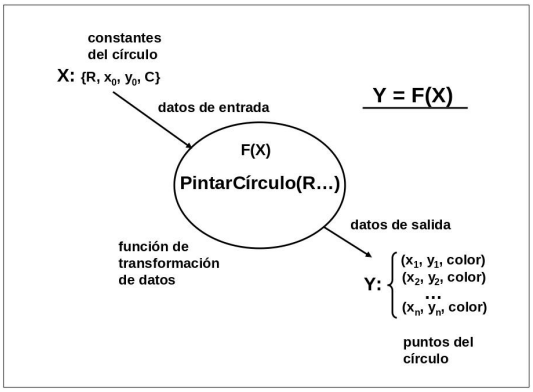


Figura 4.1: Sistema software como función de transformación de datos

A este tipo de esquema se le denomina *diagrama de flujo de datos*. **El diagrama de flujo de datos es un esquema asíncrono** (no expresa secuencias); las flechas sólo indican los flujos de datos, no el orden de ejecución.

El principal problema del enfoque estructurado es latente en el momento en el que queremos añadir más elementos e interactuar con ellos, por ejemplo, pintar varios círculos y actuar sobre los mismos de forma selectiva, digamos borrar el segundo que se pintó. Podríamos hacer un bucle para crear n círculos, pero si queremos guardarlos tendríamos que añadir tantas variables como círculos, con el objetivo de retener cada conjunto de constantes. Este sistema es una duplicación del sistema para solo un caso.

La disgregación de los conceptos en datos y funciones tiene sus pros y sus contras, por ejemplo, este enfoque permite trabajar directamente con la idea de base de datos o archivo, lo cual puede ser beneficioso. Sin embargo, esta disociación implica **disminuir nuestro nivel de abstracción**.

4.1.2. Enfoque orientado a objetos

El enfoque orientado a objetos es la forma particular de pensar el software en términos de elementos que colaboran entre sí para realizar tareas. Este enfoque nos da un nivel de abstracción superior al estructurado, asociando cada elemento del

problema a un elemento software. Cada elemento software tiene las propiedades íntegras de cada elemento del discurso (lo que *hace a una cosa ser una cosa*).

Ejemplo: Pintar un círculo

En el ejemplo anterior, definimos un objeto `círculo` que cumple las propiedades de un círculo según la definición que hemos adaptado para nuestro sistema (en este caso, el objeto contiene un centro y un radio) y tiene los mecanismos para pintarse y crearse como elemento.

El enfoque de objetos piensa:

- En variables software capaces de recordar las constantes de un círculo, capaces de pintar un círculo y capaces de crearse a sí mismas como variables.
- En el sistema software en términos de la interacción de estas variables, dadas sus respectivas capacidades para ejecutar operaciones, es decir, cómo relacionar todas las variables para conseguir que se realice la tarea de pintar círculos.

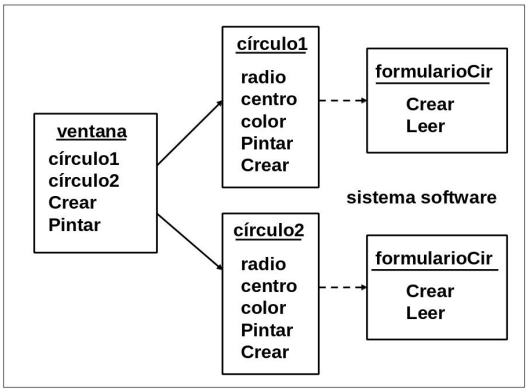


Figura 4.2: Sistema software con enfoque de objetos

Este esquema muestra el sistema software, donde se aprecian las relaciones entre las variables software que ejecutan la tarea de pintar un círculo. Como vemos, el sistema software, aun aplicando el mismo algoritmo, tiene una organización diferente, y por tanto sus propiedades también varían.

4.1.3. Objetos

Esto que hemos ido llamando *variables software* se conocen en este enfoque como **objetos**, y amplían la idea de la variable software tratada en el enfoque

estructurado, ya que tienen capacidad de expresar cualquier cosa, incluso operaciones. Otra definición complementaria de objeto es la siguiente: *Un objeto es un elemento software cualitativamente distinto capaz de expresar un concepto más amplio, más ambiguo: cosa.*

Los objetos interactúan entre ellos mediante **mensajes**, solicitudes a objetos para que ejecuten operaciones. Una buena analogía de objetos y mensajes es el teatro, los objetos son *actores* y los mensajes son *su diálogo*; el programa describe a los actores, lo que tienen que hacer y decir.

Propiedades de los objetos

Los objetos son definidos por su nombre, asociado con una dirección de la memoria de la máquina. Esta definición puede considerar además las propiedades que, generalmente, se clasifican en *atributos y operaciones*, también denominadas *métodos*.

- **Atributo:** Propiedad de un objeto, que está compuesto por un objeto o por una variable software tradicional. Los lenguajes OOP puros (como Small-Talk) solo admiten objetos como atributos, pero los lenguajes híbridos (como Java) también admiten variables software tradicionales.
- **Operación:** Propiedad de un objeto que expresa su capacidad para ejecutar la rutina indicada por la misma. Las operaciones representan cualquier código capaz de ejecutar acciones.

Los lenguajes de programación acostumbran a distinguir los atributos de las operaciones para elevar la eficiencia de compilación, pero en principio, **no hay razón para distinguirlos**.

Globalidad de los atributos y operaciones: Los atributos de un objeto son globales para todas las operaciones del mismo. Esto es, cualquier línea de código de cualquier método de un objeto tiene acceso inmediato a todos los atributos del propio objeto. Esto facilita el acceso a los atributos, pero tiene los siguientes inconvenientes:

- Cuando una operación de un objeto quiere usar otra operación del mismo, esta operación no conoce cómo usarla, ya que en la cabecera de la operación llamada no se refleja la relación con los atributos del objeto, y podría interceder en el uso de los atributos de la operación llamante.

- Al modificar alguna línea de código de una operación debemos revisar todas las líneas de código de todas las operaciones, ya que cambiar el comportamiento de una operación que a priori puede ser necesaria para el resto de operaciones podría desencadenar en un mal funcionamiento del objeto.

Visibilidad de los atributos y operaciones: Los objetos son, de una forma coloquial, *parcelas bien definidas*; accesibles desde dentro pero no tan fácilmente desde fuera. El acceso desde fuera está regulado por un çontrol de visibilidad". Los elementos designados como *públicos* son accesibles a todos los objetos del sistema software. Los *privados* son accesibles sólo desde el propio objeto.

Sobre la ambigüedad

Una de las característica más importantes del enfoque orientado a objetos es la capacidad de éstos para expresar alternativas o significados diversos, lo que comúnmente llamamos **ambigüedad**.

El enfoque estructurado utiliza dato y función de transformación de datos como elementos del sistema software, mientras el enfoque de objetos utiliza los elementos objeto, con el significado de *cosa*, y mensaje con el significado de solicitud de servicio a una variable. El significado ambiguo de *cosa* que tienen los objetos nos permite hacer un diseño estructurado con aspecto (ropaje?) de objetos, pero no al contrario (diseño de objetos con aspecto estructurado).

El enfoque de objetos se acomoda mejor a la diversidad de problemas que aborda el software. Los objetos son más tolerantes para expresar la idea general de función que el enfoque estructurado, el cual está obligado a expresar esa función en términos de "entrada, proceso y salida". No obstante, esta mayor libertad de expresión tampoco es gratis siempre. A menudo hay que aceptar cualidades "extrañas" como la capacidad de pintarse, ampliarse, moverse, borrarse, etc. que tienen los círculos software, para ajustarse al enfoque.

La **ambigüedad del enfoque de objetos** también facilita que un **elemento software**, por sí solo, **exprese completamente un concepto**, por ejemplo círculo, mientras que el enfoque estructurado obliga, muchas veces, a disociarlos en funciones y datos.

La figura muestra la evolución del software hacia el aumento de la ambigüedad de sus elementos primarios. También nos muestra claramente qué es un TAD, es decir, un *tipo abstracto de datos*. Estos TAD **no se consideran objetos**, ya que,

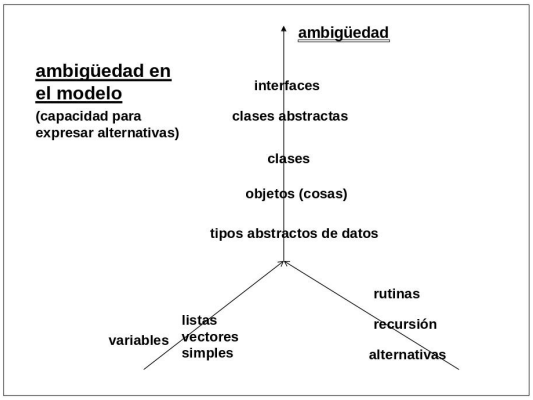


Figura 4.3: Aumento de la capacidad de ambigüedad en elementos software

a diferencia de éstos, siempre deben ser datos (un objeto no tiene por qué ser un dato, puede ser cualquier cosa, no está comprometido a desempeñar dicha función ni ninguna en particular).

Objetos vs Realidad

Gracias a la mayor capacidad expresiva de los objetos se dice a menudo que los objetos reflejan mejor la realidad. Pero esta idea perjudica, más que beneficia porque reflejar la realidad:

- No es el propósito de los sistemas software, salvo que sean de simulación.
- Dificulta los cambios en los sistemas software.
- ¿Qué significa? La realidad varía según el espectador.

Refactorización

Se llama *refactorización* a los procesos que **modifican el sistema para mejorar alguna cualidad interna sin alterar sus funciones**. A diferencia de la forma tradicional, que aspira a alcanzar un diseño perfecto, la refactorización acepta que el objetivo es funcionar a tiempo, y que después de probarlo se podrá mejorar el funcionamiento del sistema software.

Encapsulado y principio de ocultación

Como un objeto es una *parcela de software*, se suele decir que un objeto **encapsula** sus atributos y operaciones, lo cual es correcto, no distorsiona nuestra analogía con la parcela. No obstante, cuando se asocia encapsulado con ocultación

de información tenemos un problema.
Es cierto que el enfoque de objetos facilita algunas formas de desarrollo de software, como los prototipos evolutivos y el diseño en paralelo, pero no son facilidades intrínsecas.
El principio de ocultación establece que:

- Cada módulo es independiente del código de la implementación de los demás (los clientes se comunican con los servicios por medio de interfaces, que permiten realizar las operaciones sin conocimiento de la modificación del código anteriormente mencionado).
- La interfaz o definición de cada módulo debe revelar lo menos posible del trabajo interno del módulo.

4.1.4. Representación gráfica de los objetos

En este apartado veremos varios tipos de gráficos para representar objetos, sus atributos, operaciones y la interacción entre ellos (es decir, los mensajes).

UML (Unified Modelling Language)

UML es un sistema de símbolos y diagramas (un lenguaje) que sirve para expresar (modelar) los diseños de los sistemas software. La palabra unificado refleja el acuerdo de sus autores.

Objetos: UML representa a los objetos mediante **cajas**. Si se usa la definición extendida de objeto, la caja marca tres zonas: la superior contiene, subrayado, el nombre del objeto; la zona central contiene los atributos y la inferior los métodos u operaciones. En caso de usar la definición compacta de objeto, la caja solo contiene la zona superior. En UML los elementos públicos son diferenciados de los privados según el símbolo que precede su nombre; + para los públicos y - para los privados.

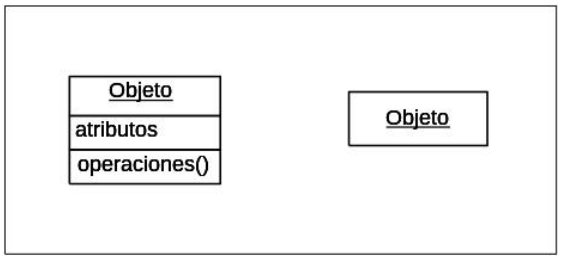


Figura 4.4: Representación de un objeto en UML

Mensajes: UML representa los mensajes mediante líneas dirigidas etiquetadas con el nombre de su operación, **desde el llamante al llamado**.

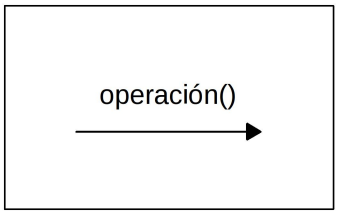


Figura 4.5: Representación de un mensaje en UML

Diagrama de secuencias: Los objetos que participan en la interacción se dibujan, horizontalmente, en la parte superior del diagrama a través de sus esquemas simplificados (cajas conteniendo sólo el nombre subrayado). Debajo de cada objeto se dibuja una línea vertical discontinua llamada línea de vida que indica, en el eje tiempo, la existencia del objeto. Los mensajes se colocan entre las líneas de vida de los objetos, siguiendo la secuencia de ejecución. Si se quiere resaltar la devolución de algún valor, se puede dibujar una flecha discontinua apuntando hacia el objeto emisor, etiquetada con el nombre del valor de retorno.

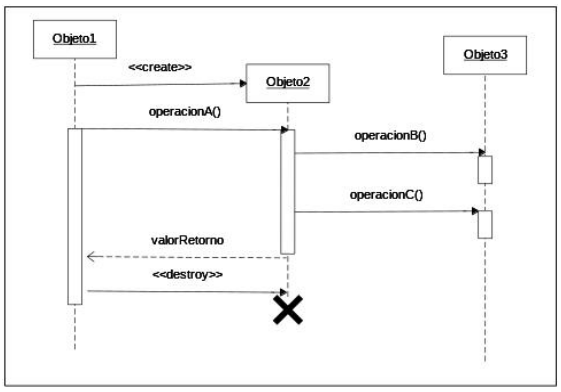


Figura 4.6: Diagrama de secuencias en UML

Para indicar la creación o destrucción de un objeto se utilizan, respectivamente, las etiquetas estereotipadas «create» y «destroy» en el mensaje que se le envía al objeto. La creación de un objeto se distingue dirigiendo el mensaje a la caja que representa al objeto. La destrucción de un objeto se muestra dibujando una X grande al final de su línea vida.

Diagrama de colaboración: Un diagrama de colabroación e sun diagrama de interacción que resalta la organización de los objetos que envían y reciben los mensajes. Este diagrama muestra un conjunto de objetos, los enlaces entre ellos y los mensajes que intercambian.
Un enlace es una instancia de una asociación o dependencia entre clases. Se representa con una línea continua que une los dos objetos.

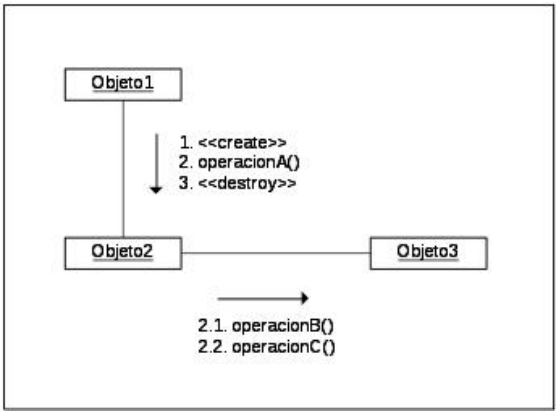


Figura 4.7: Diagrama de colaboración en UML

Los mensajes se escriben junto a los enlaces, indicando el sentido con una flecha que apunta hacia el receptor y numerándolos para expresar el orden de ejecución. El diagrama de colaboración ofrece una vista de conjunto de estructura y funcionamiento, pero incide en la estructura, afectando a la claridad del funcionamiento (toca seguir la secuencia de mensajes saltando de una línea a otra, buscando la numeración).
Además, esta forma de representación dificulta la modificación del diseño, por lo que interesa usar este diagrama cuando no interese demasiado seguir el funcionamiento y se esperen pocas modificaciones del diseño.

4.1.5. Clases

Una **clase** es una definición intensiva de un conjunto de objetos. Establece las propiedades distitntivas de cualquier elemento del conjunto que define.

Considerando las clases se podría decir que un objeto es una **instancia** de una clase o un elemento del conjunto definido por una clase. La notación UML de un objeto en particular, teniendo en cuenta la clase a la que pertenece, es `objeto:clase`. Si se quiere expresar un objeto cualquiera (anónimo) de una clase, la notación sería `:clase`.

Las clases enriquecen el enfoque de objetos. Un objeto es una cosa y una clase define un conjunto de cosas con iguales propiedades, pero valores distintos, por tanto las clases establecen una generalización, una **abstracción** mayor que los objetos. La **estructura y relaciones de los elementos del sistema** software se pueden pensar en términos de las **clases**, y dejar los objetos para pensar los diseños dinámicos particulares del sistema.

La representación de las clases y los objetos coinciden prácticamente en el lenguaje UML, con la salvedad de que el nombre de las clases no se subraya.

Diagrama de clases

El diagrama de clases expresa la estructura u organización del sistema software en términos de las clases. Además de intervenir en el funcionamiento, el diseño del diagrama es clave porque expresa la organización del sistema, y ésta decide sobre aspectos fundamentales: significado, facilidad de desarrollo en paralelo y faciidad de modificación.

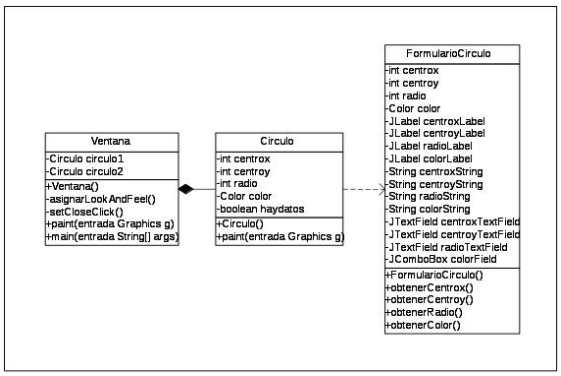


Figura 4.8: Diagrama de clases en UML

La figura muestra el diagrama de clases del sistema para dibujar círculos. Se aprecian las clases, los elementos que componen cada clase y las relaciones entre las clases, así como la visibilidad de dichos elementos.
El diagrama de clases complementa a los diagramas de secuencias, muestra la forma del soporte de los mecanismos, mientras que los diagramas de secuencias muestran el funcionamiento parcial de los mecanismos.

El diagrama de secuencias define el funcionamiento parcial del mecanismo para pintar círculos y el diagrama de clases correspondiente describe la organización de los elementos del sistema en términos de los conjuntos y sus relaciones.

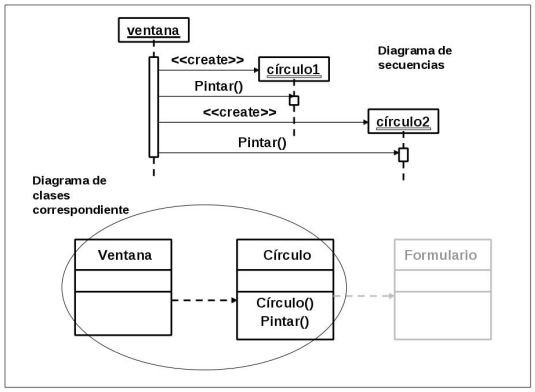


Figura 4.9: Correspondencia entre diagrama de clases y de secuencias en UML

Relaciones entre clases

El enfoque de objetos acepta cuatro tipos de relaciones entre elementos:

Dependencia: Se denomina *relación de dependencia* a la relación de una clase hacia otra cuando **los objetos de la primera usan objetos de la segunda**. Esta relación es direccional.
En UML la relación de dependencia se representa con una flecha discontinua del elemento dependiente A hacia el elemento independiente B (del que usa hacia el usado).

Asociación: Se denomina *relación de asociación* a la relación de una clase hacia otra cuando **los objetos de la primera contienen atributos que son objetos de la segunda clase**.
En UML la asociación unidireccional se representa con una flecha continua de la clase que contiene a la clase cuyos objetos son contenidos. La asociación bidireccional se representa con una línea continua entre las clases asociadas.

Agregación: Se denomina *relación de agregación* entre clases cuando hay una **relación "todo/parte" entre los objetos de ambas clases**.
En UML se utiliza una línea continua y un rombo hueco en el lado del continente para representar una relación de agregación.
La relación de agregación introduce conceptualmente una estructura jerárquica del tipo 'está formado por' en el diagrama de clases, lo cual la diferencia de las relaciones anteriormente mencionadas (que son entre iguales).

Composición: La *composición* es otra forma de agregación con una fuerte relación de pertenencia: Un objeto "parte" sólo pertenece a un objeto "todo" y cuando se destruye el objeto "todo" se destruye el o los objetos "parte".
En la composición el objeto "todo" controla la disposición y vida de las partes. Hay un cierto efecto transitivo: lo que le ocurre al todo le debe ocurrir a las partes. En UML se utiliza una línea continua y un rombo relleno en el lado del continente.

Herencia

Desde el punto de vista formal, se establece que:
La *herencia* es una relación entre clases donde una de ellas comparte la estructura o comportamiento definido en una (herencia simple) o más clases (herencia múltiple). Las subclases pueden compartir atributos y operaciones de sus superclases, y también redefinir algunos de estos elementos.
Tenemos que tener mucho cuidado con la herencia, ya que, como ya hemos visto, generar dependencias fuertes es malo para el mantenimiento y la modificación de nuestro diseño software.
En un UML, la herencia se denota siempre desde la subclase a la superclase, con un triángulo hueco a la superclase.

Polimorfismo

El polimorfismo es una de las cualidades más importantes del enfoque de objetos por su aporte a la ambigüedad en el diseño. Se asocia con el mecanismo de herencia y permite que la operación definida con la misma cabecera sea implementada de maneras distintas. Formalmente, se denomina polimorfismo a la **capacidad de una operación para manifestar un comportamiento diferente dependiendo del objeto que la ejecuta.**

Principio de sustitución de Liskov

Después del análisis de los muchos problemas de la herencia, el principio de sustitución de Liskov, formulado hace casi dos décadas, ofrece un camino útil y confiable para aprovechar los favores de la herencia. Literalmente el principio define, en términos de una sustitución segura, cuando una subclase es un subtipo de una superclase.
"Si para cada objeto O1 de tipo S hay un objeto O2 de tipo T tal que para todos los programas P definidos en términos de T, el comportamiento [interno] de P no cambia cuando O1 es sustituido por O2, entonces S es un subtipo de T."
Pero lo interesante es ver el principio de sustitución desde otra perspectiva. *La herencia permite sustituir un objeto por otro, es decir cambiar una tarea por otra, sin riesgo, siempre que las subclases sean subtipos de las superclases.*

El cumplimiento del principio de sustitución exige que los métodos de las clases derivadas deban mantener las siguientes relaciones con los métodos de la clase base:

- La clase derivada debe tener un método correspondiente a cada método de la clase base. Este método puede heredarse directamente de la clase base o sobrescribirse.
- Cada método de la clase derivada que se corresponda a un método de la clase base debe requerir lo mismo o menos que la clase base. Es decir, si se sobrescribe un método heredado de la clase base, las precondiciones del método deben ser más débiles o permisivas que las del método de la clase base. El nuevo método no puede ser más restrictivo que el método heredado.
- Cada método de la clase derivada que se corresponda a un método de la clase base debe garantizar lo mismo o más que la clase base. Es decir, si se sobreescribe un método heredado de la clase base, las postcondiciones del método de las clases derivada deben ser más fuertes o rigurosas que las heredadas de la clase base. Dicho de otro modo, el método de la clase derivada no debe comprometerse a ofrecer mayores resultados o resultados diferentes; sólo debe comprometerse a hacer lo que hace el método de la clase base, garantizando también las propiedades adicionales. Por ejemplo, si un método de la clase base devuelve un número mayor que el argumento que recibe, un método de una clase derivada podría devolver un número primo mayor que el argumento. Pero no estaría permitido que el método de la clase derivada devolviese un número menor o igual que el argumento.
- Está permitido que la clase derivada introduzca nuevos métodos adicionales que no aparezcan en la clase base.

Clases abstractas

Se denomina *método abstracto* al método que sólo expresa la cabecera y carece de código interno. Dicho de otro modo, un método que está declarado pero no implementado.
Se denomina *clase abstracta* a la clase que contiene al menos un método abstracto porque refleja la abstracción de ese método. Por ejemplo, la clase Figura. También se les llama clases virtuales o diferidas.

Bloque 5

Arquitectura

5.1. Introducción a la arquitectura

Requerimientos →Diseño →Construcción →Pruebas →Implantación.

La arquitectura de Software de un sistema es el **conjunto** de **estructuras** necesarias para **razonar** sobre el sistema. Relaciona distintos elementos de software como son los objetos o los hilos de ejecución; el modelo del sistema, los diagramas, la lógica; o las entidades físicas como los nodos donde se ejecutará Software.

Desde una perspectiva de alto nivel, la arquitectura de Software cubre diferentes componentes del sistema. Tendrá en cuenta los requerimientos y fines del sistema, pero a la vez la creación del modelo, las dependencias y los escenarios de uso. Es decir, la arquitectura de software maneja sin encargarse de la implementación final, los aspectos técnicos y de uso que ocurrirán durante el desarrollo del sistema. Crea por tanto una estructura orientada al rendimiento, usabilidad y modificabilidad (**requisitos de calidad**).

5.2. Requerimientos

Objectivos de negocio→Drivers arquitectónicos →Decisiones arquitectura→ Arquitectura documentada→Riesgos Deudas

Un requerimientos es una **especificación** que describe alguna funcionalidad, atributo o factor de calidad de un sistema software.

Requerimiento→Diseño→Documentación →Evaluación→Implementación

Existe una amalgama de intereses y requerimientos entre los distintos actores que usarán el sistema. Si bien todos juegan un papel fundamental, a nivel de equipo

de desarrollo se deberán satisfacer los requerimientos funcionales, es decir, usar una *combobox* para elegir los billetes. [**?**, p. 14]

La **ISO 9126** ofrece una descripción de los criterios de calidad del software (sección 1.5):

- Funcionalidad
- Confiabilidad
- Usabilidad
- Eficiencia
- Mantenibilidad
- Portabilidad

Los **drivers** son un subconjunto de requerimientos que definen la estructura de un sistema. Existen los drivers **funcionales**, **de atributos de alta calidad**, y los drivers de **restricciones**.

Funcionales Descomposición del sistema. Relevancia y complejidad.

Calidad Los atributos de calidad.

Restricciones Técnicas y de gestión.

Métodos para identificar drivers arquitectónicos

Existen diferentes métodos para identificar drivers arquitectónicos. Podemos basarnos en *talleres de atributos*, métodos de diseño o *FURPS*.

El propósito de los talleres de atributos es ayudar a elegir la arquitectura adecuada para un sistema de Software. El modelo *QAW* (Talleres de calidad del Atributo) se centra en los requisitos del cliente, y no hace necesaria la existencia previa de una arquitectura software.

5.3. Diseño de estructuras

El diseño es la especificacione de **objeto**, creado por algún **agente**, que busca alcanzar ciertos **objectivos**, en un **entorno** particular, usando un conjunto de **componentes** básicos, satisfaciendo una serie de **requerimientos** y sujetándose a determinadas **restricciones**.

Teniendo en cuenta lo que nos han pedido, juntar piezas que tenemos teniendo en cuenta nuestras restricciones para describir lo que queremos hacer.
Arquitectura→Interfaces→Detalle de los módulos

Se diseña en base a los principios de **modularidad**, **alta cohesión y bajo acomplamiento** y de **mantener las cosas simples**.

Los patrones de diseño juegan un papel fundamental en la especiación de los drivers. Se abstraen problemas ya resueltos sin llegar a representar soluciones detalladas para luego adaptarlo a cada caso particular. Cuando los diseños son más concretos, se llegan a crear elementos software reutilizables que proporcionan la funcionalidad genérica enfocándose a la resolución de un problema específico. Así nacen los **frameworks**.

A la hora de diseñan las **interfaces** se identifican los mensajes que se intercambian.

5.4. Diseño de Arquitecturas

El problema del diseño de la arquitectura se resuelve mediante diseños **basados en atributos**, **centrados en arqutectura** o con **vistas y perspectivas**. El método de **Rozansky & Woods**.

	ADD	**ACDM**	**Rozansky & Woods**
Mecánica y enfoque	Diseño iterativo descomponiendo elementos recursivamente	Iteraciones de diseño, documentación y evaluación.	Iteraciones de diseño, documentación y evaluación.
Participantes	Arquitecto	Arquitecto y otros	Arquitecto y otros
Entradas	Drives	Drivers y alcance	Vistas
Salidas	Esbozos de vistas	Vistas	Vistas
Criterios de terminación	Se satisfacen los drivers	Los experimentos no revelan riesgos o son aceptables	Los interesados están de acerudo en que el diseño satisface sus preocupaciones.
Conceptos de diseño utilizados	Técnicas y patrones	Estilos arquitectónicos, patrones y prácticas	Estilos arquitectónicos y patrones.

Figura 5.1: Comparación de métodos de dieseño de arquitecturas
Interesante ver la figura 1.1

5.5. Documentación

Generación de documentos que describen las estructuras de la arquitectura con el propósito de comunicar efectivamente a los interesados en el sistema.

La documentación se apoya en vistas para la descripción de las estructuras. Se componen de un diagrama que representa los objetos de la estructura y de información textual que ayuda a comprender el diagrama.

La **vista lógica** representa en el diagrama *unidades* de implementación, que pueden ser en base a la funcionalidad o la responsabilidad.

Otras *vistas* son las de **comportamiento**, las **físicas** o la de Windows™ [1].

5.6. Evaluación

La evaluación es la técnica para evitar que los defectos lleguen a los usaurios finales o que se presenten en momentos donde corregirlos sea complicado.

La evaluación sirve para determinar si el software cumple con los criterios de calidad (1.3). Al evaluar un sistema se pueden producir *desviaciones* respecto a las necesidades de los usuarios o respecto a la construcción correcta del producto. Al evaluar las arquitecturas se busca satisfacer los drivers arquitectónicos (5.2).

5.7. Implementación

La implementación busca generar diseños detallados de los módulos y otros elementos siempre de acuerdo con la arquitectura. Se ajustan los diseños y errores, pero no se cambia la arquitectura.

- Diseñar la estructura del sistema basándose en la arquitectura.
- Basarse en los requisitos funcionales (5.2).
- Desarrollar[2].

La resolución de las desviaciones (*errores*) se resuelve mediante controles de calidad en los que se **verifica el código**, el **diseño**. Además de realizar **pruebas** y **auditorías**.

[1] Que no nos gusta.

[2] Picar código y fixes.

5.8. Arquitectura Software

Componente Bloque del sistema. Parte que combinas con la arquitectura.

Servicio Funcionalidad que los componentes proporcionan a los actores.

Al dividir un sistema en componente, hay que definir los servicios que proporciona cada componente.

5.8.1. Estilos

El estilo es la forma general de un sistema, semejante a lo que serían los patrones de diseño (5.3). Al definir un estilo, se deben especificar los elementos como los bloques básicos de contrucción, las conexiones entre los bloques y las reglas que espeficican cómo se combinan los servicios.

Técnica	**Patrón**
Abstracción	Niveles
Encapsulación	Expedidor-receptor
Ocultación de información	Reflexión, Composite
Modularización	Niveles, Pipes & Filters, Composite
Acoplamiento y cohesión	Publicador-Suscriptor,Cliente-Despachador-Servidor
Separación de intereses	Modelo-Vista-Controlador

Figura 5.2: Patrones que ayuan a aplicar técnicas

5.8.2. Índice de un documento de arquitectura

- Objetivos
- Requerimientos (*funcionales, no funcionales*) (5.2)
- Decisiones y justificación
- Modelo conceptual (5.5)
  - Modelo de componentes lógicos
  - Modelo de procesos
  - Modelo físico
  - Modelo de despliegue
- Despliegue de la arquitectura

Otra información relevante del documento de arquitectura es presentar distintos diagramas:

- Diagrama de Clases (*Lógica*).
- Diagrama de Paquetes (*Desarrollo*).
- Diagrama de Interacción (*Procesos*).
- Diagrama de Despliegue (*Física*).

Pasos en la identificación de un problema

Metas del proceso→Recogida de información→Conceptos de la Arquitectura→Cliente de la solución→Definición del problema

5.8.3. Patrones

1. Especificar el problema.
    - Dividir el problema.
    - Encontrar el contexto.
    - Considerar pros/cons.
    - Acceder al catálogo de patrones.
2. Seleccionar la categoría de los patrones (*arquitectónicos o de diseño*).
3. Categoría del problema.
4. Comparar descripciones del problema.
5. Comparar beneficios y compromiso.
6. Elegir la mejor variante.

Entre los ejemplos de patrones están: *N-Niveles, Filtros y Tuberías, Pizarra, Modelo-Vista-Controlador.*

Índice alfabético