Tema 5

Diseño y programación orientada a objetos. Elementos y componentes software: objetos, clases, herencia, métodos, sobrecarga. Ventajas e inconvenientes. Patrones de diseño y lenguaje de modelado unificado (UML).

Guion-resumen

1. **Diseño y programación orientada a objetos**
   - 1.1. Diseño orientado a objetos
2. **Elementos y componentes software: objetos, clases, herencia, métodos, sobrecarga**
   - 2.1. Evolución de los objetos
   - 2.2. La programación orientada a objetos (POO-OOP)
   - 2.3. La abstracción
   - 2.4. El encapsulado
   - 2.5. La herencia
   - 2.6. El polimorfismo
   - 2.7. Clase
   - 2.8. Objeto
   - 2.9. Propiedades y métodos
   - 2.10. Mensajes
   - 2.11. Identidad
   - 2.12. Reutilización o reusabilidad
   - 2.13. Jerarquía
   - 2.14. Concurrencia, persistencia y tipificado
   - 2.15. Modularidad
   - 2.16. Relaciones entre los conceptos asociados al modelo de objetos
3. **Ventajas e inconvenientes**
   - 3.1. Ventajas de la POO
   - 3.2. Inconvenientes de la POO
4. **Patrones de diseño y lenguaje de modelado unificado (UML)**
   - 4.1. UML
   - 4.2. Diagramas

1. Diseño y programación orientada a objetos

La programación orientada a objetos y todos los lenguajes que la usan (entre los cuales destacaremos: C++, C # y JAVA) han cobrado gran renombre, no quizás tanto por su funcionalidad, sino por la revolución que causaron en la forma de pensar el programador y que con gran seguridad marcarán los lenguajes del futuro.

La idea es tan sencilla como: si quiero hacer miles de bicicletas qué sería mejor:

1. ¿Ir haciendo "una a una" hasta alcanzar la cantidad prevista?
2. ¿Crear un molde (**clase**) para hacer bicicletas (**objetos**) y a partir de ese molde crear las bicicletas?

Está claro, la segunda opción es la ideal. Es más, si nos paramos a pensar: todas las bicicletas que salgan del molde serán exactamente iguales a no ser que modifiquemos el molde. A las bicicletas que quiera una vez creadas les pudo acoplar los atributos que se desee. Si le ponemos un atributo nuevo al molde todas las nuevas bicicletas tendrán ese nuevo atributo. Podríamos seguir citando ventajas durante folios y folios de este libro, pero lo que se debe es comprender la idea.

El término **objeto** surgió a principios de los sesenta en varios campos de la informática, para referirse a nociones que eran diferentes en su apariencia, pero relacionadas entre sí.

Cada concepto que usamos los humanos es una idea particular o una comprensión de nuestro mundo, los conceptos adquiridos nos permiten sentir y razonar acerca de las cosas en el mundo. A estas cosas a las que se aplican nuestros conceptos se las llama objetos. Un objeto puede ser real (ejemplo: una piedra, un avión) o abstracto (ejemplo: organización). De manera formal decimos: un objeto es cualquier cosa, real o abstracta, acerca de la cual almacenamos datos y los métodos que controlan dichos datos.

Un objeto puede estar compuesto por otros objetos; estos últimos, a su vez, pueden estar compuestos de objetos, del mismo modo que una máquina está formada por partes y éstas, también, están formadas por otras partes. Esta estructura intrincada de los objetos permite definir objetos muy complejos. Las técnicas orientadas a objetos permiten que el software se construya a partir de objetos de comportamiento específico.

El elemento fundamental de la OOP-POO (OOP en inglés, POO en español) es, como su nombre lo indica, el objeto. Podemos definir un **objeto (en programación)** como un **conjunto complejo de datos y programas que poseen estructura y forman parte de una organización**. Esta definición especifica varias propiedades importantes de los objetos. En primer lugar, un objeto no es un dato simple, sino que contiene en su interior cierto número de componentes bien estructurados. En segundo lugar, cada objeto no es un ente aislado, sino que forma parte de una organización jerárquica o de otro tipo.

Un objeto puede considerarse como una especie de cápsula dividida en tres partes:

— Las **relaciones** permiten que el objeto se inserte en la organización y están formadas esencialmente por punteros a otros objetos.

— Las **propiedades** distinguen un objeto determinado de los restantes que forman parte de la misma organización y tiene valores que dependen de la propiedad de que se trate. Las propiedades de un objeto pueden ser heredadas a sus descendientes en la organización.

— Los **métodos** son las operaciones que pueden realizarse sobre el objeto, que normalmente estarán incorporados en código que el objeto es capaz de ejecutar y que también pone a disposición de sus descendientes a través de la herencia.

1.1. Diseño orientado a objetos

1.1.1. Introducción

El proceso de desarrollo de software es aquel en el que las necesidades del usuario son traducidas en requisitos de software, estos transformados en diseño y el diseño implementado en código. Podemos definir el diseño en el software como el proceso de aplicar distintas técnicas y principios con el propósito de definir un producto con los suficientes detalles como para permitir su realización física. Con el diseño se pretende construir un sistema que satisfaga determinada especificación del sistema, se ajuste a las limitaciones impuestas por el medio de destino y respete requisitos sobre forma, rendimiento, utilización de recursos...

A través del diseño producimos un modelo o representación técnica del software que se va a desarrollar. El diseño es uno de los procesos básicos sobre el que se asienta la calidad del software. Se trata de un proceso iterativo a través del cual se traducen los requisitos en una representación del software. Se representa a un alto nivel de abstracción, un nivel que se puede seguir hasta requisitos específicos de datos, funcionales y de comportamiento.

1.1.2. Metodologías de diseño

— **Diseño de datos.** Modelo de información a estructuras de datos.

— **Diseño arquitectónico.** Define las relaciones entre los elementos estructurales del programa.

— **Diseño procedimental.** Se transforman los elementos estructurales del programa en una descripción procedimental del software.

— **Diseño de interfaz.** Describe cómo se comunica el software consigo mismo y con su entorno.

1.1.3. Directrices de diseño

— El diseño debe implementar todos los requisitos explícitos contenidos en el modelo de análisis y debe acomodar todos los requisitos implícitos que desee el cliente.

— El diseño debe ser una guía que puedan leer y entender los que construyan el código y los que prueban y mantienen el software.

— El diseño debería proporcionar una completa idea de lo que es el software, enfocando los dominios de datos, funcional y de comportamiento desde la perspectiva de la implementación.

1.1.4. Principios básicos de diseño

— El diseñador debe considerar enfoques alternativos juzgando a cada uno en relación a los requisitos del problema, los resultados disponibles y los criterios de calidad interna.

— Se deben seguir los pasos de diseño hasta el modelo de análisis.

— El diseño no va a reinventar nada que ya esté inventado.

— El diseño debería presentar uniformidad de integración.

— Debe estructurarse para admitir cambios.

— El diseño no es escribir código y escribir código no es diseñar.

— Se debería valorar la calidad del diseño mientras se crea, no después de terminado.

1.1.5. Patrones de diseño

Un patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada varias veces sin hacerlo dos veces de la misma forma. Un patrón aborda un problema de diseño recurrente que aparece en situaciones específicas de diseño y presenta una solución para este. Los patrones identifican y especifican abstracciones que están por encima del nivel de las clases e instancias, o de componentes.

Un patrón tiene cuatro elementos esenciales:

— **El nombre del patrón** se usa para describir un problema de diseño, sus soluciones y consecuencias en una o dos palabras.

— **El problema** describe cuándo aplicar el patrón. Explica el problema y su contexto. Podría describir problemas específicos de diseño del tipo de cómo representar algoritmos como objetos. Podría describir estructuras de clases u objetos que son síntomas de un diseño inflexible.

— **La solución** describe los elementos que construyen el diseño, sus relaciones, responsabilidades y colaboraciones. El patrón proporciona una descripción abstracta de un problema de diseño y cómo una disposición general de elementos lo resuelve.

— **Las consecuencias** son los resultados e inconvenientes de aplicar el patrón. Aunque las consecuencias se ignoran cuando describimos las decisiones de diseño, son críticas para evaluar las alternativas de diseño y para entender los costes y beneficios de aplicar el patrón.

1.1.6. Tipos de patrones

A) Patrones de creación

Los patrones de creación conciernen al proceso de creación de objetos. Los patrones de creación proporcionan ayuda a la hora de crear objetos, principalmente cuando esta creación requiere tomar decisiones. Esta toma de decisiones puede ser dinámica. Estos patrones ayudan a estructurar y encapsular estas decisiones. En algunas ocasiones existe más de un patrón que se puede aplicar a la misma situación. En otras ocasiones se pueden combinar múltiples patrones convenientemente. Un patrón de creación asociado a clases usa la herencia para variar la clase que se instancia, mientras que un patrón de creación asociado a objetos delegará la instanciación a otro objeto.

Hay dos formas de clasificar los patrones de creación basándose en las clases de objetos que se crean. Una es clasificar las clases que crean los objetos (Factory Method), otra forma está relacionada con la composición de objetos; definir un objeto que es responsable de conocer las clases de los objetos producto, en esta característica se apoyan los patrones Abstract Factory, Builder o Prototype.

— **Factory method** proporciona una interfaz para crear un objeto, pero deja a las subclases decidir qué clase instanciar. Permite a una clase delegar la instanciación a las subclases.

— **Abstract Factory** proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar su clase concreta.

— **Builder** separa la construcción de un objeto complejo de su representación para que el mismo proceso de construcción pueda crear diferentes representaciones.

— **Prototype** especifica el tipo de objetos a crear usando una instancia prototipo, y crea nuevos objetos copiando este prototipo.

— **Singleton** asegura que una clase solo tiene una instancia, y proporciona un punto de acceso global a esta.

B) Patrones estructurales

Tratan de la composición de clases y objetos. Los patrones estructurales están relacionados con cómo las clases y los objetos se combinan para dar lugar a estructuras más complejas. Puede hacerse aquí la misma distinción que hacíamos en los patrones de creación y hablar de patrones estructurales asociados a clases (Adapter) y asociados a objetos (Bridge, Composite, Decorator, Facade, Flyweight, Proxy), los primeros utilizarán la herencia, los segundos la composición.

Los patrones estructurales asociados con objetos describen formas de componer los objetos para conseguir nueva funcionalidad. La flexibilidad de la composición de objetos viene de la posibilidad de cambiar la composición en tiempo de ejecución, lo que es imposible con la composición estática de clases.

— **Adapter** convierte la interfaz de una clase en otra interfaz que espera el cliente. Permite trabajar juntas a clases que de otra forma no podrían hacerlo por incompatibilidad de "interfaces".

— **Bridge** desacopla una abstracción de su implementación para que los dos puedan variar independientemente.

— **Composite** compone objetos en estructuras de árbol para representar jerarquías parte-todo. Permite a los usuarios tratar objetos individuales y composiciones de manera uniforme.

— **Decorator** agrega responsabilidades adicionales a un objeto dinámicamente. Proporcionan una alternativa flexible a las subclases para extender funcionalidad.

— **Facade** proporciona una interfaz unificada a un conjunto de interfaces en un subsistema. Define una interfaz de alto nivel que hace el subsistema más fácil de usar.

— **Flyweight** comparte para proporcionar un gran número de objetos pequeños eficientemente.

— **Proxy** proporciona un sustituto para otro objeto, para controlar el acceso a él.

C) Patrones de comportamiento

Los de comportamiento caracterizan las maneras en las que las clases u objetos interactúan y se distribuyen las responsabilidades. Estos patrones de diseño están relacionados con algoritmos y asignación de responsabilidades a los objetos. Los patrones de comportamiento describen no sólamente patrones de objetos o clases sino también patrones de comunicación entre ellos. Nuevamente se pueden clasificar en función de que trabajen con clases (Template Method, Interpreter) u objetos (Chain of Responsability, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor).

La variación de la encapsulación es la base de muchos patrones de comportamiento. Cuando un aspecto de un programa cambia frecuentemente, estos patrones definen un objeto que encapsula dicho aspecto. Los patrones definen una clase abstracta que describe la encapsulación del objeto.

— **Observer** define una dependencia uno a varios entre objetos de manera que cuando un objeto cambia su estado, todos los objetos dependientes son notificados y actualizados automáticamente.

— **Mediator** define un objeto que encapsula cómo un conjunto de objetos interactúan. Promueve bajo acoplamiento evitando que los

objetos se refieran entre ellos explícitamente, y permite variar su interacción independientemente.

— **Chain of Responsibility** evita el acoplamiento del emisor de una petición a su receptor dando a más de un objeto una oportunidad para manejar la petición. Encadena los objetos receptores y pasa la petición a lo largo de la cadena hasta que un objeto la maneja.

— **Template Method** define un esqueleto de un algoritmo en una operación, aplazando algunos pasos a las subclases. Permite a las subclases redefinir ciertos pasos de un algoritmo sin cambiar la estructura del mismo.

— **Interpreter,** dado un lenguaje, define una representación para su gramática y un intérprete que usa la representación para interpretar las sentencias en el lenguaje.

— **Strategy** define una familia de algoritmos, encapsulados individualmente, y los hace intercambiables. Permite a un algoritmo cambiar independientemente de los usuarios que lo usen.

— **Visitor** representa una operación que va a ejecutarse sobre elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos con los que opera.

— **State** permite a un objeto alterar su comportamiento cuando cambia su estado interno. El objeto aparenta cambiar su clase.

— **Command** encapsula una petición como objeto, por lo tanto permite valorar o analizar los usuarios con diferentes peticiones, colas o peticiones de conexión, y permite operaciones que se pueden deshacer.

— **Iterator** proporciona un medio para acceder a los elementos de un objeto agregado secuencialmente sin exponer su representación subyacente.

El ámbito especifica cuándo los patrones se aplican principalmente a las clases o a los objetos. Los patrones de clase tratan con relaciones entre las clases y sus subclases. Estas relaciones se establecen a través de la herencia, así que son estáticas, fijas en tiempo de compilación. Los patrones de objetos tratan con las relaciones de objetos, que pueden cambiar en tiempo de ejecución y son más dinámicos. Casi todos los patrones utilizan la herencia en algún punto.

Los patrones de clase de creación delegan alguna parte de la creación a subclases, mientras que los de objeto delegan en otro objeto. Los estructurales de clases usan la herencia para componer las clases, mientras que los de objeto describen maneras de reunir objetos. Los patrones de clase de comportamiento usan herencia para describir algoritmos y flujo de control, mientras que los de objeto describen cómo un grupo de objetos cooperan para realizar una tarea que no podría llevar a cabo un objeto solo.

Los métodos de diseño orientados a objetos favorecen varios enfoques. Se puede escribir el problema y extraer nombres y verbos como clases y ope-

raciones. Se puede modelar desde el punto de vista de las colaboraciones y responsabilidades en el sistema. Se puede modelar el mundo real y trasladar los objetos encontrados en el análisis al diseño. Los patrones de diseño ayudan a identificar las abstracciones menos obvias y los objetos que pueden capturarlas.

D) Interfaces

Cada operación declarada por un objeto especifica el nombre de una operación, los objetos que toma como parámetros y lo que devuelve. A esto se le llama signatura. Al conjunto de todas las signaturas definidas por las operaciones de un objeto se le llama la interfaz del objeto. La interfaz de un objeto caracteriza el conjunto completo de peticiones que se le pueden mandar a un objeto. Cualquier petición que coincida con una signatura de la interfaz de un objeto puede ser mandada a ese objeto.

Un tipo es el nombre que se suele usar para denotar una interfaz determinada. Un objeto puede tener varios tipos, y objetos muy diferentes pueden ser del mismo tipo. Dos objetos del mismo tipo solo necesitan compartir parte de sus interfaces. Las interfaces pueden contener otras interfaces como subconjuntos. Se dice que un tipo es un subtipo de otro si su interfaz contiene la interfaz de su supertipo (herencia).

Los "interfaces" son fundamentales en los sistemas orientados a objetos. Los objetos solo son conocidos a través de sus interfaces. No hay manera de saber nada acerca de un objeto o de pedirle que haga algo sin pasar a través de su interfaz. La interfaz de un objeto no dice nada acerca de su implementación, pues diferentes objetos pueden implementar las peticiones de manera diferente.

Cuando una petición se envía a un objeto, la operación que se realiza depende tanto de la petición como del objeto que la recibe. La asociación en tiempo de ejecución de una petición a un objeto y la operación se llama **ligadura dinámica**.

La ligadura dinámica significa que hacer una petición no se asocia a una implementación particular hasta el tiempo de ejecución. Por lo tanto, se pueden escribir programas que esperen un objeto con una interfaz particular, sabiendo que cualquier objeto que tenga la interfaz correcta aceptará la petición. Es más, la ligadura dinámica permite sustituir objetos que tienen interfaces idénticas entre sí en tiempo de ejecución (polimorfismo). El polimorfismo simplifica las definiciones de los usuarios, desacopla los objetos entre sí y permite que varíen sus relaciones en tiempo de ejecución.

Los patrones de diseño ayudan a definir interfaces identificando sus elementos principales y los tipo de datos que pueden ser enviados a través de una interfaz. Un patrón de diseño puede indicar también qué es lo que no se debe poner en una interfaz. El patrón memento describe cómo encapsular y salvar el estado interno de un objeto para que este pueda ser restaurado posteriormente. El patrón estipula que los objetos memento deben definir dos interfaces: una restringida para los usuarios, y otra privilegiada para que únicamente el objeto original pueda guardar y recuperar su estado.

Los patrones de diseño también especifican las relaciones entre "interfaces". En particular, muchas veces necesitan que algunas clases tengan "interfaces" similares, o añaden restricciones a las interfaces de algunas clases.

Existen dos beneficios al manipular los objetos en términos de su interfaz definida por una clase abstracta:

— Los usuarios ignoran en tipo específico de objeto que están usando, mientras que respondan a la interfaz que el cliente espera.

— Los usuarios ignoran las clases que implementan los objetos. Únicamente deben conocer la clase abstracta que define el interfaz.

Esto reduce significativamente las dependencias de implementación entre los subsistemas. Los patrones de creación abstraen el proceso de creación, de manera que ofrecen diferentes maneras de asociar una interfaz con su implementación de manera transparente a la instanciación. Los patrones de creación aseguran que el sistema se escribe en términos de interfaces, no implementaciones.

1.1.7. Herencia y composición

Las dos maneras de reutilizar la funcionalidad en los sistemas orientados a objetos son la herencia de clases (caja blanca) y la composición de objetos (caja negra). Cada una tiene sus ventajas e inconvenientes.

— **Ventajas:** La herencia de clases se define estáticamente en tiempo de compilación, y se usa directamente, ya que está proporcionada por el lenguaje de programación. La herencia permite modificar de manera más sencilla la implementación que se está reutilizando. Si se sobrescriben algunos, pero no todos los métodos, podría ocurrir que los no sobrescritos llamen a los sobrescritos.

— **Inconvenientes:** No se puede cambiar la implementación heredada de las clases base en tiempo de ejecución, ya que se define en tiempo de compilación. La herencia rompe el encapsulamiento. Las superclases definen parte de la representación física de las subclases. Las dependencias de implementación pueden causar problemas cuando se intenta reutilizar una subclase... se podría solucionar heredando de interfaces.

La composición de objetos se define dinámicamente en tiempo de ejecución mediante objetos que toman referencias a otros objetos. La composición requiere que los objetos respeten sus interfaces, lo que requiere interfaces diseñados cuidadosamente. A cambio, no se rompe el encapsulamiento, y cualquier objeto puede ser reemplazado por otro en tiempo de ejecución siempre que tenga el mismo tipo. Existen menos dependencias de implementación.

La composición de objetos tiene otro efecto sobre el diseño de sistemas. Cada clase se mantiene encapsulada y enfocada en una tarea. Las clases y jerarquías de clases permanecerán pequeñas en lugar de crecer hasta convertirse en

monstruos inmanejables. Por otra parte un diseño basado en la composición de objetos tendrá más objetos (y menos clases).

Idealmente, se debería poder conseguir toda la funcionalidad que se necesite ensamblando componentes existentes mediante composición de objetos. Pero este caso no se suele dar, porque el conjunto de componentes nunca es suficientemente rico. La reutilización mediante la herencia hace más fácil componer nuevos componentes con los antiguos. De esta manera la herencia y la composición trabajan juntas.

1.1.8. Delegación

Sirve para simular la herencia mediante la composición. La principal ventaja de la delegación es que hace fácil componer comportamientos en tiempo de ejecución y cambiar la manera en la que están compuestos. Las desventajas con que es más difícil de entender y las ineficiencias de tiempo de ejecución. Varios patrones usan delegación: state, strategy y visitor. La delegación es un ejemplo extremo de composición de objetos. Nos muestra que siempre se puede reemplazar herencia por composición de objetos como mecanismo de reutilización de código.

1.1.9. Diseño

La llave para maximizar la reutilización está en anticiparse a los nuevos requisitos y cambios a los requisitos existentes, y diseñar el sistema para que pueda evolucionar como corresponde.

Los cambios pueden provocar redefinición de las clases, reimplementación, modificación del cliente y nuevos tests. El rediseño afecta a muchas partes del sistema software.

Los patrones de diseño nos ayudan a evitarlo asegurando que el sistema puede cambiar en ciertos aspectos determinados. Cada patrón de diseño permite que algún aspecto de la estructura del sistema varíe independientemente de otros aspectos, por lo tanto hace el sistema más robusto para un tipo determinado de cambio.

Se listan a continuación las causas de rediseño más comunes junto con los patrones de diseño que las abordan:

1. **Crear un objeto especificando una clase explícitamente.** Especificar el nombre de una clase cuando se crea un objeto, nos compromete a una implementación particular en lugar de a una interfaz. Este compromiso puede complicar cambios futuros. Para evitarlo, los objetos se deben crear indirectamente.

   Patrones de diseño: Factoría abstracta, factory method, prototipo.

2. **Dependencias de operaciones específicas.** Cuando se especifica una operación particular, nos comprometemos a una manera de satisfacer una petición. Evitando peticiones especificadas en tiempo de compilación, se hace más fácil de cambiar la manera en la que se satisface una petición.

   Patrones de diseño: Cadena de responsabilidad, comando.

3. **Dependencia de la plataforma hardware y software.** Las APIs de los sistemas operativos son diferentes para las distintas plataformas. El software que depende de una plataforma particular será más difícil de portar a otras plataformas. Podría incluso ser difícil de mantener actualizada en su plataforma nativa. Por lo tanto, es importante diseñar el software para limitar las dependencias de la plataforma.

   Patrones de diseño: factoría abstracta, puente.

4. **Dependencia de las representaciones o implementaciones de objetos.** Los usuarios que saben cómo se representa, almacena o implementa podrían necesitar ser cambiados cuando cambien los objetos. Ocultar esta información de los usuarios evita cambios en cascada.

   Patrones de diseño: factoría abstracta, puente, memento, proxy.

5. **Dependencias algorítmicas.** Los algoritmos se extienden, optimizan y se reemplazan habitualmente durante el desarrollo y la reutilización. Los objetos que dependen de un algoritmo tendrán que cambiar cuando el algoritmo cambie. Por lo tanto los algoritmos que tengan pinta de cambiar deben ser aislados.

   Patrones de diseño: builder, iterator, estrategia, template method, visitor.

6. **Acoplamiento fuerte.** Las clases que están muy acopladas son difíciles de reutilizar individualmente, ya que dependen unas de otras. El acoplamiento fuerte lleva a sistemas monolíticos, donde no se puede cambiar o eliminar una clase sin entender y cambiar muchas otras clases. El sistema se convierte en una masa densa que es difícil de aprender, portar y mantener. El acoplamiento débil incrementa la probabilidad de que una clase pueda ser reutilizada por sí misma y que el sistema pueda ser aprendido, portado, modificado y extendido más fácilmente.

   Patrones de diseño: factoría abstracta, puente, cadena de responsabilidad, comando, fachada, mediador, observador.

7. **Extender la funcionalidad mediante la herencia.** Adaptar un objeto mediante la herencia muchas veces no es fácil. Cada nueva clase tiene una implementación fija por encima (inicialización, terminación...). Definir una subclase además requiere un entendimiento en profundidad de la clase base. Por ejemplo, sobrescribir una operación puede requerir sobrescribir otra. Una operación sobrescrita podría ser necesitada para llamar a una operación heredada. Y la herencia puede llevar a una explosión de clases, porque podrían necesitarse muchas subclases nuevas para una simple extensión. La composición de objetos en general y la delegación en particular proporciona una alternativa flexible a la herencia para combinar comportamientos. La nueva funcionalidad puede añadirse a una aplicación componiendo objetos existentes de nuevas maneras en lugar de definir nuevas subclases de clases existentes. Por otra parte, demasiado uso de la composición puede hacer los diseños más difíciles de entender. Muchos patrones producen diseños en los que se pueden introducir nuevas funcionalidades simplemente definiendo una subclase y componiendo las instancias con otras existentes.

Patrones de diseño: puente, cadena de responsabilidad, composite, decorador, observador, estrategia.

8. **Incapacidad para alterar las clases convenientemente.** A veces hay que modificar una clase que no puede ser modificada convenientemente. Quizás se necesita el código fuente y no se tiene. O puede que cualquier cambio necesite modificar un montón de clases existentes. Los patrones nos ayudan a modificar las clases en estas circunstancias.

   Patrones de diseño: adaptador, decorador, visitor.

2. Elementos y componentes software: objetos, clases, herencia, métodos, sobrecarga

2.1. Evolución de los objetos

Las ideas básicas sobre los objetos nacen a principios de los años setenta en la universidad de Noruega, donde un equipo dirigido por el Dr. Nygaard se dedicaba a desarrollar sistemas informáticos para realizar simulaciones de sistemas físicos. Debido a que eran programas muy complejos y el mantenimiento era muy necesario (para que el software se adaptara a nuevas necesidades), se dieron cuenta de las limitaciones de la ingeniería de software tradicional, para solucionar este problema idearon una forma de diseñar el programa paralelamente al objeto físico, donde cada componente del objeto físico se correspondía con un componente de software, con lo que se simplificaba el programa y, por tanto, el mantenimiento exigía menor esfuerzo.

Lo anterior trajo consigo otro beneficio como es la reutilización del código, hecho que por sí mismo repercute en una baja en el costo del software y en el tiempo requerido para el desarrollo de sistemas.

El primer lenguaje que implementó estas ideas fue el lenguaje SIMULA-67. Luego, en la década de los 70, XEROX en sus laboratorios de Palo Alto desarrolla SMALL-TALK.

En los años 80 tomando ideas de Simula y de Small-Talk, en los laboratorios Bell de ATT, Stroustrup crea el lenguaje C++ como sucesor del Lenguaje C, y a este se debe la gran extensión de los conceptos de objetos. En el área de la inteligencia artificial, se desarrolla Clos, una variante de Lisp orientada a objetos.

En sistemas operativos, el Next-Step de Sun es un Sistema Operativo Orientado a objetos. Microsoft trabaja en Cairo, IBM y Apple trabajan en Pink, como sistemas operativos que incluyen conceptos de objetos.

En las bases de datos, tenemos al SNAP *(Strategic Networed Applications Plataform)*, conocido en español como Sistemas distribuidos en línea orientados a objetos. Poco más tarde (1993-4) Sun crea JAVA, el cual arrasa hasta la actualidad en aplicaciones cliente y en programas ejecutables para Internet (applets).

Los conceptos de objetos entran en profundidad poco a poco en todos los ámbitos de la computación, y de la misma manera que ha sido inevitable aprender programación estructurada o bases de datos relacionales, ahora se hace necesario aprender programación orientada a objetos. Los conceptos de análisis, diseño y programación con objetos son fáciles de dominar una vez que se tiene una base de programación en un lenguaje. Para poder construirnos nuestras propias librerías de clases o para llegar a ser un programador de objetos de alto rendimiento, se requiere un poco mas de práctica. En el mercado existen bibliotecas de clases, y también varios lenguajes traen bibliotecas de clases que le permiten al programador realizar ciertas tareas sin tener que programarlas.

No obstante, no debemos confundir Programación Orientada a Objetos, donde el programador puede usar clases precreadas o crearlas el mismo, con Programación Basada en Objetos donde las clases ya están precreadas y el programador solo puede usarlas o modificarlas. Ejemplos de lenguajes de Programación Basada en Objetos son: Clipper que tiene objetos ya creados como el Tbrowse que permite el manejo de tablas y Visual Basic, al igual que Delphi tienen objetos como botones o cuadros de diálogo con los que permite desarrollar interfaces de usuario con un mínimo de programación.

Nota. *Hemos de destacar que Visual Basic .NET es también un lenguaje de última generación Orientado a Objetos.*

2.2. La programación orientada a objetos (POO-*OOP*)

La idea principal de POO es construir programas que utilizan objetos de software. Un objeto puede considerarse como una entidad independiente de cómputo con sus propios datos y programación. En computadoras modernas, las ventanas, los menús y las carpetas de archivos, por ejemplo, suelen representarse con objetos de software. Pero los objetos pueden aplicarse a muchos tipos de programas Se incluirían datos que describen los atributos físicos, y programación (métodos), que gobierna la manera en que funciona internamente y en que interactúa con otras partes relacionadas.

En contraste, la programación tradicional trabajaba con bytes, variables, matrices, índices y otros utensilios de programación que resultaba difícil relacionar con el problema actual. Además, la programación tradicional se concentra en los procedimientos paso a paso, llamados algoritmos, para realizar las tareas deseadas. Por esta razón, a la programación tradicional también se le conoce como programación orientada a procedimientos.

Todos los lenguajes de programación están formados por dos elementos: código y datos. Cuando los programas empezaron a hacerse complicados y su código es enorme y casi inmanejable, se pensó crear una nueva forma de pensar de la cual surgió la programación orientada a objetos. En ella un programa se organiza en torno a sus datos (objetos) y a un conjunto de interfaces bien definidas para esos datos.

La POO es un modelo de programación que utiliza objetos ligados mediante mensajes para la resolución de problemas. La idea inicial siempre ha sido organizar los programas a imagen y semejanza de la organización de los objetos en el mundo real.

Las técnicas orientadas a objetos se basan en organizar el software como una colección de objetos discretos que incorporan tanto estructuras de datos como comportamiento. Esto contrasta con la programación convencional, en la que las estructuras de datos y el comportamiento estaban escasamente relacionadas.

Las características principales del enfoque orientado a objetos serán estudiadas con detenimiento en este tema y son:

— Abstracción.

— Encapsulación.

— Herencia.

— Polimorfismo.

— Clase.

— Objeto.

— Método.

— Mensajes.

— Identidad.

— Reutilización.

— Jerarquía.

— Concurrencia.

— Modularidad.

A partir de estos elementos fundamentales, trataremos de dar un enfoque tanto estructurado como orientado a objetos.

Un objeto en el mundo real tiene una apariencia, peso, volumen y se puede definir por la función que realiza. Tiene por tanto un conjunto de características (atributos) que describen su naturaleza y funcionalidad. Un objeto es cualquier cosa, real o abstracta, en la cual almacenamos datos y los métodos que controlan dichos datos, pongamos la vista enfrente y estaremos rodeados de objetos.

La teoría de los objetos puede ser aplicada a cualquier sistema, por que la organización de estos (objetos) es lo que define al sistema, ya que este posee **atributos** y características individuales que lo hacen organizacional, desde su denominación en sí, su **clase**, su **poliformismo**, el proceso de **encapsulación**, la **herencia** que sucede a otro objeto, los **mensajes** que de algún modo llevan a cabo que se realice una operación, el **método** como se hace, su identidad, que lo difiere de los demás, su **reutilización**, el **orden jerarquizado**, su **abstracción**, **modularidad** y, por ultimo, su **concurrencia.**

La solución para tratar con la complejidad típica de los programas es **"Divide y vencerás":**

— **Descomposición Algorítmica** (top-down o estructural): rompe el sistema en partes, cada una representando un pequeño paso del proceso. Métodos de diseño estructurados conducen a descomposiciones algorítmicas, donde la atención se centra en el flujo del sistema.

— **Descomposición orientada a objetos:** trata de identificar semánticamente el dominio del problema. El entorno del problema se estudia como un conjunto de agentes autónomos (objetos) que colaboran para realizar un comportamiento complejo.

Algorítmica.	Orientada a objetos.
Diagramas tipo árbol.	Varias posibilidades.
Desmenuza el problema.	Identifica semánticamente el problema.
Se programa en detalle.	Se programa a lo grande.
Lenguajes imperativos.	Lenguajes declarativos.

En cuanto a las características de las descomposiciones algorítmica y orientada a objetos hemos de diferenciar tres términos usados en la orientación a objetos:

— **Análisis orientado a objetos:** es un método de análisis que examina los requerimientos desde la perspectiva de clases y objetos encontrada en el vocabulario original del problema.

— **Diseño orientado a objetos:** es un método de diseño que abarca el proceso de descomposición orientado a objetos y una notación para describir modelos lógicos y físicos, dinámicos y estáticos, del sistema bajo diseño.

— **Programación orientada a objetos:** es el método de implementación en el cual los programas se organizan como colecciones cooperantes de objetos, cada uno de los cuales representa un ejemplo de alguna clase, y cuyas clases son todas miembros de una jerarquía de clases unidas por relaciones.

El concepto renovador de la tecnología de POO es la anexión de procedimientos de programas a elementos de datos. Esta idea cambia la separación tradicional entre datos y programas. A esta nueva unión se le llama **encapsulamiento** y el resultado es un *objeto de software*. En JAVA, por ejemplo, todos los procedimientos están encapsulados y se les llama **métodos**. Por ejemplo, un objeto de ventana en un sistema de interfaz gráfica del usuario contiene las dimensiones físicas de la ventana, la ubicación en la pantalla, los colores de primer plano y de fondo, los estilos de borde y otros datos relevantes. **Encapsulados** junto con estos **datos**, se encuentran los métodos

para mover y modificar el tamaño de la propia ventana, para cambiar sus colores, para desplegar texto, para reducirlo a un icono, etc. Otras partes del programa de interfaz del usuario solo llaman a un objeto de ventana para realizar estas tareas enviándole **mensajes** bien definidos. El trabajo de un objeto de ventana consiste en realizar las acciones apropiadas y mantener actualizados sus datos internos. Para programas fuera del objeto no importa mucho la manera exacta en que se realizan estas tareas ni las estructuras de los datos internos. La interfaz pública formada por los diferentes tipos de mensajes que envía un objeto, definen por completo la manera de usarlo. Esta es la interfaz de programación de aplicaciones (**API**, *Application Programming Interface*) del objeto. El ocultamiento de detalles internos hace que un objeto sea **abstracto**.

La separación entre la interfaz pública y el funcionamiento interno no es difícil de comprender. Por ejemplo, cuando montamos en un coche, pensamos en un medio de transporte que nos lleve de un lugar a otro. No pensamos en un amasijo de hierros, plástico, etc. Además, si montamos en el coche del vecino sabremos conducirlo igualmente pues su forma de utilización es la misma. Su funcionamiento interno lo dejamos para cuando falla y tenemos que pagarle al taller. Cuando se ejecuta un programa, los objetos se crean, los mensajes se envían y los objetos se destruyen. Estas son las únicas operaciones permitidas sobre ellos. Los datos o los métodos internos (privados) de un objeto están fuera de los límites del público. El desacoplamiento de los mecanismos privados de los objetos de las rutinas externas a ellos, reducen en gran medida la complejidad de un programa.

En POO se define una clase para cada tipo diferente de objeto. Se utilizan una definición de clase y valores iniciales apropiados para crear una **instancia** (objeto) de la clase. A esta operación se le conoce como instanciación de objetos.

La tecnología de POO necesita formas fáciles para construir objetos sobre otros, para eso existen dos métodos principales, **composición y herencia**. El primero permite que objetos existentes se utilicen como componentes para construir otros. Por ejemplo, un objeto de calculadora puede estar compuesto por otro de unidad aritmética y uno más de interfaz de usuario. La herencia es una función importante de POO que le permite ampliar y modificar clases existentes sin cambiar su código.

Una subclase hereda código de su superclase y, también, agrega sus propios datos y métodos. La herencia permite la extracción de elementos comunes entre objetos similares o relacionados. También permite que se utilicen clases de bibliotecas de software para muchos propósitos diferentes o no previsibles. Heredar de una clase se conoce como herencia simple y heredar de varias clases se conoce como herencia múltiple.

Además, POO permite el **polimorfismo**, que es la capacidad de un programa para trabajar con diferentes objetos. Se permite la creación de objetos compatibles que son transferibles. La modificación y el mejoramiento de un programa polimórfico puede ser tan solo cuestión de enlazar objetos actualizados.

Una **clase** es como el plano de los **objetos**. Describe las estructuras de datos del objeto y sus operaciones asociadas. Una vez que se ha definido una clase, es posible declarar los objetos que le pertenecen y utilizarlos en un pro-

grama. Por lo general una clase contiene miembros que pueden ser **campos y métodos**. Los primeros son variables que almacenan datos y objetos. Los segundos son funciones que codifican operaciones. Es así que ellos reciben argumentos, realizan cálculos predefinidos y devuelven resultados.

Un **mensaje** enviado a un objeto invoca un método de ese objeto, le pasa argumentos y obtiene el valor que devuelve. Los objetos interactúan al enviar y recibir mensajes.

Una **clase** proporciona el nombre bajo el que se reúnen los miembros para formar una unidad de cálculo que puede operar con independencia de otras partes del programa. Con **objetos**, puede construirse un programa grande con muchas unidades pequeñas, independientes y que interactúan entre sí. La orientación a objetos puede reducir significativamente la complejidad del programa, aumentar su flexibilidad y mejorar las posibilidades de volver a usarlo. Un programa puede definir sus propias clases, utilizar las precreadas (generalmente guardadas en bibliotecas de clases) y emplear las que han sido creadas por otros programadores. Las clases pueden estar organizadas en paquetes con un nombre. Cada **paquete** puede contener uno o más archivos de código fuente.

Veamos a continuación cada uno de los conceptos clave de la programación orientada a objetos en profundidad.

2.3. La abstracción

La abstracción es una de las bases de la POO. Desde siempre el programador ha intentado abstraerse y no ver el programa como un conjunto complejo de código. Se pretender ignorar los detalles y obtener una visión en su conjunto. Una forma de obtener una buena abstracción es utilizando **clasificaciones jerárquicas.** En las clasificaciones jerárquicas primero vemos el sistema desde su exterior. Luego, en un segundo nivel, nos vamos adentrando en él y viendo los subsistemas, en un tercer nivel nos adentramos en cada uno de los subsistemas, etc.

Ejemplo. *Un camión:*

Si nos abstraemos lo vemos exteriormente como un objeto de grandes dimensiones y con gran capacidad de carga que puede transportar de un lugar a otro.

El camión lo podemos dividir en cabina, remolque, usos, coste, etc. (subsistemas).

Dentro de la cabina vemos que es un habitáculo donde el conductor se acomoda y controla el camión.

A su vez dentro del subsistema cabina tenemos: los asientos, la radio, controles, etc. Cada uno de ellos con una función delimitada. Dentro de cada uno de ellos, ...

Esta **abstracción jerárquica** de sistemas en subsistemas se puede aplicar a los programas que el programador crea y de los datos tradicionales obtenemos su **abstracción en objetos**. Cada uno de estos objetos tiene un comportamiento y funcionalidad propio, que se pueden tratar como entidades inde-

pendientes y que responden a **mensajes** (secuencia de pasos de un proceso) que les dicen lo que tienen que hacer y en qué orden.

Una abstracción denota las características esenciales de un objeto que lo distinguen de todos los demás tipos de objetos, y proporciona así fronteras conceptuales definidas con nitidez, desde la perspectiva del observador. Todo objeto es único. Sin embargo, la abstracción elimina algunas distinciones para que podamos ver los aspectos comunes entre los objetos.

La abstracción es una de las vías fundamentales por la que los humanos podamos combatir la complejidad. Una abstracción se centra en la visión externa de un objeto y, por lo tanto, sirve para separar el comportamiento esencial de un objeto de su implantación.

Sin la abstracción solo sabríamos que cada objeto es diferente de los demás, con ella se omiten de manera selectiva varias características distintivas de uno o más objetos, lo que permite concentrarnos en las características que comparten. Para hacerlo más entendible, diremos que la **abstracción**: es el **acto o resultado de eliminar diferencias entre los objetos, de modo que podamos ver los aspectos más comunes.**

La abstracción denota las características esenciales que distinguen a un objeto de otros tipos de objetos, definiendo precisas fronteras conceptuales, relativas al observador. Las características de la abstracción son:

— Surge del reconocimiento de similitudes entre ciertos objetos, situaciones o procesos en el mundo real.

— Decide concentrarse en estas similitudes e ignorar las diferencias.

— Enfatiza detalles con significado para el usuario, suprimiendo aquellos detalles que, por el momento, son irrelevantes o distraen de lo esencial.

— Deben seguir el "principio de mínimo compromiso", que significa que la interfaz de un objeto provee su comportamiento esencial, y nada más que eso. Pero también el "principio de mínimo asombro": capturar el comportamiento sin ofrecer sorpresas o efectos laterales.

2.4. El encapsulado

Se trata de un mecanismo que permite juntar el código y los datos y que mantiene a ambos alejados de posibles usos indebidos. Para ello el acceso al código y a los datos se realiza de forma controlada a través de una **interfaz** bien definida. El encapsulado permite que se realice la migración de las implementaciones tras el paso del tiempo, sin necesidad de reescribir de nuevo todo el código (reutilización o reusibilidad).

Ejemplo. *Estás en tu coche. Si pisas el freno lo que debe hacer el coche es frenar y no debe activarse el parabrisas, ni acelerar, ni aumentar el volumen de la radio, etc. Eso sucede gracias a que el sistema de frenado está perfectamente definido y funciona como un sistema independiente y con una función*

muy definida. Puede comunicarse con otros sistemas como las luces para que se activen las luces de frenado, pero esta comunicación está perfectamente definida y aunque no funcionen las luces el coche ha de frenar igualmente. Igualmente, si el coche no frena el fallo es del sistema de frenado y no de otro sistema.

Con el encapsulado lo único que debemos conocer del sistema es cómo acceder a él, sin preocuparnos de los detalles internos (abstracción) y estamos seguros de que no se van a producir efectos secundarios imprevistos.

Por tanto, el encapsulado consiste en ocultar los detalles de instrumentación de un objeto, a la vez que se provee de una **interfaz pública** por medio de sus operaciones permitidas. En los modelos orientados a objetos, lo que realmente nos importa es el comportamiento de los objetos y no cómo está instrumentado ese comportamiento. Así, si la instrumentación de un objeto cambia pero su interfaz se mantiene igual, los objetos que interactúan con él no se verán afectados por esos cambios. Además, el encapsulamiento oculta la complejidad de la instrumentación. **El encapsulamiento es el proceso de compartimentar los elementos de una abstracción que constituyen su estructura y su compartamiento.**

Dicho de otro modo, **cada objeto es una estructura compleja en cuyo interior hay datos y código, todos ellos relacionados entre sí, como si estuvieran encerrados conjuntamente en una cápsula**. Es, a esta propiedad, a la que se denomina encapsulamiento. Sirve para separar la interfaz contractual de una abstracción y su implantación. El hecho de que cada objeto sea una cápsula facilita enormemente que un objeto determinado pueda ser transportado a otro punto de la organización, o incluso a otra organización totalmente diferente que precise de él. Si el objeto ha sido bien construido, sus métodos seguirán funcionando en el nuevo entorno sin problemas. Esta cualidad hace que la POO sea muy apta para la reutilización de programas.

El encapsulamiento es importante porque separa el comportamiento del objeto de su implantación. Esto permite la modificación de la implantación del objeto sin que se tengan que modificar las aplicaciones que lo utilizan. La encapsulación sirve para separar la interface de una abstracción y su implementación.

— Es un concepto complementario al de abstracción.

— La encapsulación esconde la implementación del objeto que no contribuye a sus características esenciales.

— La encapsulación da lugar a que las clases se dividan en dos partes:

1. **Interface:** captura la visión externa de una clase, abarcando la abstracción del comportamiento común a los ejemplos de esa clase.

2. **Implementación:** comprende la representación de la abstracción, así como los mecanismos que conducen al comportamiento deseado.

— Se conoce también como ocultamiento o privacidad de la información.

Como hemos visto, cada objeto es una estructura compleja en cuyo interior hay datos y programas, todos ellos relacionados entre sí, como si estuvieran encerrados conjuntamente en una cápsula. Esta propiedad (encapsulamiento), es una de las características fundamentales en la OOP.

No obstante, los objetos son inaccesibles, e impiden que otros objetos, los usuarios, o incluso los programadores conozcan cómo está distribuida la información o qué información hay disponible. Esta propiedad de los objetos se denomina **ocultación de la información**. Esto no quiere decir, sin embargo, que sea imposible conocer lo necesario respecto a un objeto y lo que contiene. Si así fuera no se podría hacer gran cosa con él. Lo que sucede es que las peticiones de información a un objeto deben realizarse a través de mensajes dirigidos a él, con la orden de realizar la operación pertinente. La respuesta a estas órdenes será la información requerida, siempre que el objeto considere que quien envía el mensaje está autorizado para obtenerla.

El hecho de que cada objeto sea una cápsula facilita enormemente que un objeto determinado pueda ser transportado a otro punto de la organización, o incluso a otra organización totalmente diferente que precise de él. Si el objeto ha sido bien construido, sus métodos seguirán funcionando en el nuevo entorno sin problemas. Esta cualidad hace que la POO sea muy apta para la reutilización de programas.

Otra forma de ver su definición: una de las características centrales de POO es la partición del programa completo en pequeñas entidades autónomas, llamadas objetos, con interacciones bien definidas. Esta característica reduce significativamente la complejidad global y ayuda a la calidad del programa de diferentes maneras. Un objeto organiza datos y operaciones relacionadas en una caja negra, que oculta las estructuras de datos, las representaciones y el código interno de la vista exterior. Una estructura de datos es concreta cuando se conocen sus elementos exactos. Los métodos de programación tradicional dependen mucho de datos concretos. Una estructura de datos es abstracta si solo conocemos su comportamiento y no sus detalles de implantación. Por tanto, la abstracción de datos pone el énfasis en el ocultamiento de los detalles internos de los datos y solo presenta su comportamiento externo. Por ejemplo, sin conocer los detalles de la construcción de un automóvil, podemos manejarlo con solo conocer comportamientos como: “al girar el volante en el sentido de las manecillas del reloj se gira a la derecha”. Esto deja la implantación de la conducción a la caja negra, que puede utilizar una de las varias alternativas de dirección: normal, hidráulica, de engranajes, etc. Además, un objeto también contiene mecanismos o códigos que son necesarios para operar las estructuras de datos que están adjuntos a los códigos para formar una unidad inseparable. A esta técnica se le llama **encapsulación**.

En POO el comportamiento externo de un objeto es un contrato de interfaz entre el objeto y sus clientes (los programas que utilizan el objeto). Este convenio contiene datos/operaciones que quedan disponibles para los clientes externos a partir de un objeto y que documenta su significado preciso.

2.5. La herencia

La herencia es el proceso mediante el cual un objeto adquiere las propiedades de otro. Gracias a la herencia se consigue llevar a cabo la idea vista anteriormente de **clasificación jerárquica.** Si una clase dada encapsula algunos atributos entonces cualquier clase hija de la anterior (subclase) heredará los atributos de la anterior más los que se le quieran definir.

La herencia nos da reusabilidad de código. El concepto de herencia se refiere a la comparición de atributos y operaciones basadas en una relación jerárquica entre varias clases. Una clase pude definirse de forma general y luego redefinirse en sucesivas subclases. Cada clase hereda todas las propiedades de sus superclases y añade sus propiedades particulares.

Ejemplo 1. *Los seres vivos tienen una serie de propiedades. Los animales al ser seres vivos heredan esas propiedades a las cuales añaden otras. Los mamíferos, al ser animales, heredan todos los atributos de los animales a las cuales añaden otras. Los humanos, como animales, etc.*

Ejemplo 2. *Imagínate que tienes un molde para hacer bicicletas sin marchas. Has hecho miles de bicicletas y todo va bien. De repente deseas hacer bicicletas con marchas. ¿Qué te parece mas apropiado: crear un nuevo molde (es decir, crear de nuevo con todo lo que ello conlleva), o coger el molde anterior y acoplarle las marchas? Esta es la idea de la herencia, a partir de un sistema creado se crea uno nuevo acoplándole nuevos atributos.*

La herencia es el medio por el cual los objetos de una clase pueden acceder a variables y funciones miembro contenidas en una clase previamente definida, sin tener que volver a realizar esas definiciones. Existen dos tipos de herencia:

A) Herencia simple

La herencia simple es aquella en la que una clase puede heredar la estructura de datos y operaciones de una superclase. Es una relación entre clases en la que una clase comparte la estructura o el comportamiento definido en otra.

B) Herencia múltiple

La herencia múltiple se da cuando una clase puede heredar la estructura de datos y operaciones de más de una superclase. Es la relación entre clases en la que una clase comparte la estructura de más de una clase base.

La herencia múltiple presenta una gran dificultad y es el hecho de que puede heredar dos operaciones con el mismo nombre. Esto hace que las colisiones pueden introducir ambigüedad en el comportamiento de la subclase que hereda en forma múltiple.

Nota. *El lenguaje C++ permite la herencia múltiple, pero JAVA no lo permite.*

Es una de las propiedades más destacadas de la POO, se puede definir como un mecanismo que define nuevos objetos con base en los existentes. Lenguajes de programación como C++ y JAVA la soportan con extensión de clase, definiendo una nueva clase con base en otra ya existente sin modificarla. A la primera se le llama **subclase** o clase extendida, recibe a los miembros de una **superclase** y agrega otros de su propiedad. Aunque también es posible que la herede dado el surgimiento de jerarquías.

JAVA soporta herencia sencilla a través de extensión de clase. En este tipo de herencia una clase se extiende a partir de una sola superclase. Por ejemplo, una subclase puede extenderse de la superclase. Algunos lenguajes POO, como C++, soportan herencia múltiple porque una subclase puede tener varias superclases. Al anular la herencia múltiple, JAVA evita las dificultades y complicaciones relacionadas con este tipo de herencia.

Si una clase se designa final no puede extenderse. El uso experto de la herencia a través de subclases contribuye en gran medida a un programa bien diseñado.

Varias son las **ventajas de la herencia**, entre las cuales citamos:

— Fácil modificación de código: evita la modificación del código existente y utiliza la herencia para agregar nuevas características o cambiar características existentes.

— Reutilización de código existente: usando la herencia como base de código que funciona y está probado podemos crear nuevas clases fácilmente.

— Adaptación de programas para trabajar en situaciones similares pero diferentes: evita que se vuelvan a escribir programas muy similares porque la aplicación, el sistema de computadora, el formato de datos o el modo de operación es un poco diferente.

— Extracción de elementos comunes de clases diferentes: evita que se dupliquen códigos; y estructuras idénticas o similares de clases diferentes. Se extraen las partes comunes para formar otra clase y permita que las demás las hereden.

— Organización de objetos en jerarquías: forma grupos de objetos que tienen una **relación**. Las agrupaciones le dan una mejor organización a un programa y permite que los objetos de la misma jerarquía se utilicen como tipos compatibles, en oposición a los que carecen totalmente de relación.

2.6. El polimorfismo

El polimorfismo, que en griego significa “muchas formas”, es una característica que le permite a una interfaz ser utilizada por una clase general de acciones. La frase “una interfaz, muchas formas” resume esta idea. El polimorfismo permite que se cree un código limpio, sensible, legible y resistente.

Ejemplo. *Para abrir un archivo en Windows se pincha dos veces en su icono. Para ver el contenido de una carpeta también se pincha dos veces sobre su icono. Es decir, en ambos casos hemos pinchado dos veces para ejecutar la acción correspondiente. La interfaz ha sido la misma aunque con dos formas diferentes.*

Este concepto reduce la complejidad permitiendo que la misma interfaz se utilice para especificar una clase general de acción. Es el compilador o el intérprete el que debe seleccionar la acción específica a ejecutar.

El polimorfismo se define como la posibilidad de asumir varias formas. Permite que una misma operación pueda llevarse a cabo de varias formas, en clases diferentes. Desde este punto de vista, representa un concepto de teoría de tipos en el que un solo nombre puede denotar objetos de muchas clases diferentes que se relacionan por alguna superclase común. Cualquier objeto denotado por este nombre es, por lo tanto, capaz de responder a algún conjunto común de operaciones. Una operación es una acción o transformación que realiza o padece un objeto. La implementación específica de una operación determinada a una clase determinada se denomina método. Aunque los métodos sean distintos, llevan a cabo el mismo propósito operativo, y así estaríamos hablando también, de **polimorfismo**.

Según lo dicho, una operación es una abstracción de un comportamiento similar (pero no idéntico) en diferentes clases de objetos. La semántica de la operación debe ser la misma para todas las clases. Sin embargo, cada método concreto seguirá unos pasos específicos.

Existe el polimorfismo cuando interactúan las características de la herencia y el enlace dinámico. Esta es quizás la característica más importante de los lenguajes orientados a objeto,s después de su capacidad para soportar la abstracción, y es lo que distingue la programación orientada a objetos de otra programación más tradicional con tipos abstractos de datos. El polimorfismo es también un concepto central en el diseño orientado a objetos.

Una de las ventajas del polimorfismo es que se puede hacer una solicitud de una operación sin conocer el **método** que debe ser llamado. No es otra cosa que la posibilidad de construir varios métodos con el mismo nombre, pero con relación a la clase a la que pertenece cada uno, con comportamientos diferentes. Esto conlleva la habilidad de enviar un mismo mensaje a objetos de clases diferentes. Estos objetos recibirían el mismo mensaje global pero responderían a él de formas diferentes.

El polimorfismo, el encapsulado y la herencia son quizás los tres principios clave de la programación orientada a objetos. Cuando se aplican conjunta y adecuadamente producen un entorno de programación que admite el desarrollo de programas robustos y más fáciles de ampliar que los modelos tradicionales de diseño orientado al proceso. Permiten la reutilización de código y un menor coste de programación.

2.7. Clase

La clase es el núcleo de la POO. Se trata de una construcción lógica sobre la que se construye la orientación a objetos. En la clase se define la forma y la naturaleza de un objeto. **Una clase define un nuevo tipo de dato que se utiliza para crear objetos de ese tipo**. Dicho de otra forma: una clase es el molde a partir del cual se fabrican objetos.

Una clase está formada por **miembros de datos que a su vez son variables de instancia (datos) y métodos**. Una clase es, por tanto, una agrupación de datos (variables o campos) y de funciones (métodos) que operan sobre esos datos. A estos datos y funciones pertenecientes a una clase se les denomina variables y métodos o funciones miembro. La programación orientada a objetos se basa en la programación de clases. Un programa se construye a partir de un conjunto de clases.

Una vez definida e implementada una clase, es posible declarar elementos de esta. **Los elementos declarados de una clase se denominan objetos de la clase**. Se pueden declarar o crear numerosos objetos, la clase es lo genérico: es el patrón o modelo para crear objetos.

El cuerpo de la clase, encerrado entre { }, es la lista de atributos (variables) y métodos (funciones) que constituyen la clase. No es obligatorio, pero en general se listan primero los atributos y luego los métodos.

La definición de una clase se realiza en la siguiente forma:

[ ] class nombredelaclase

[datos] {

[lista_de_atributos]

[lista_de_métodos]

}

El esqueleto de cualquier aplicación JAVA se basa en la definición de una clase. Todos los datos básicos, como los enteros, se deben declarar en las clases antes de hacer uso de ellos.

En la práctica son pocas las sentencias que se pueden colocar fuera del bloque de una clase. La palabra clave import (equivalente al #include) puede colocarse al principio de un fichero, fuera del bloque de la clase. Sin embargo, el compilador reemplazará esa sentencia con el contenido del fichero que se indique, que consistirá, como es de suponer, en más clases.

Los tipos de clases que podemos definir son:

— **Abstract**. Una clase abstract tiene al menos un método abstracto. Una clase abstracta no se instancia (no se puede crear ningún objeto de ella), sino que se utiliza como clase base para la herencia.

— **Final**. Una clase final se declara como la clase que termina una cadena de herencia. No se puede heredar de una clase final. Por ejemplo, la clase Math es una clase final.

— **Public**. Las clases public son accesibles desde otras clases, bien sea directamente o por herencia. Son accesibles dentro del mismo

paquete en el que se han declarado. Para acceder desde otros paquetes, primero tienen que ser importadas.

— **Synchronizable**. Este modificador especifica que todos los métodos definidos en la clase son sincronizados, es decir, que no se puede acceder al mismo tiempo a ellos desde distintos threads; el sistema se encarga de colocar los flags necesarios para evitarlo. Este mecanismo hace que desde threads diferentes se puedan modificar las mismas variables sin que haya problemas de que se sobrescriban.

Citaremos ahora algunas de las normas de las clases en JAVA:

1. Todas las variables y funciones de JAVA deben pertenecer a una clase. No hay variables y funciones globales.

2. Si una clase deriva de otra (extends), hereda todas sus variables y métodos.

3. JAVA tiene una jerarquía de clases estándar de la que pueden derivar las clases que crean los usuarios.

4. Una clase solo puede heredar de una única clase (en JAVA no hay herencia múltiple). Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de Object. La clase Object es la base de toda la jerarquía de clases de JAVA. En C++ sí se permite la herencia múltiple.

5. En un fichero se pueden definir varias clases, pero en un fichero no puede haber más que una clase public. Este fichero se debe llamar como la clase public que contiene con extensión *.java. Con algunas excepciones, lo habitual es escribir una sola clase por fichero.

6. Si una clase contenida en un fichero no es public, no es necesario que el fichero se llame como la clase.

7. Los métodos de una clase pueden referirse de modo global al objeto de esa clase al que se aplican por medio de la referencia this.

8. Las clases se pueden agrupar en “packages”, introduciendo una línea al comienzo del fichero (package packageName;).Esta agrupación en packages está relacionada con la jerarquía de directorios y ficheros en las que se guardan las clases.

El término clase se refiere a la implantación en software de un tipo de objeto. Especifica una estructura de datos y los métodos operativos permisibles que se aplican a cada uno de los objetos. Una clase puede tener sus propios métodos y estructura de datos, así como también heredarlos de su superclase. La superclase es la clase de la cual hereda otra clase, llamada esta última subclase inmediata.

Una clase es una abstracción de un conjunto posiblemente infinito de objetos individuales. Cada uno de estos objetos se dice que es una instancia o ejemplar de dicha clase. Cada instancia de una clase posee sus propios valores para sus atributos, pero comparte el nombre de estos atributos y las opera-

ciones con el resto de instancias de su clase. La elección de clases es arbitraria y depende del dominio del problema.

Nota. *La industria utiliza el término clase para hacer referencia a las implantaciones de los tipos de objetos.*

Se construyen clases a partir de otras clases, las cuales a su vez se integran mediante clases. Así como los bienes manufacturados se fabrican a partir de una serie de materiales de partes y subpartes ya existentes, también el software se crea mediante una serie de materiales de clases ya existentes y probadas.

Una clase es un conjunto de objetos que comparten una estructura y comportamiento comunes.

— Clase representa una abstracción, la esencia que comparten los objetos.

— Un objeto es un ejemplo de una clase.

— Un objeto no es una clase y una clase no es un objeto (aunque puede serlo, p.e. en Smalltalk).

— Las clases actúan como intermediarias entre una abstracción y los clientes que pretenden utilizar la abstracción. De esta forma, la clase muestra:

1. Visión externa de comportamiento (interfaz), que enfatiza la abstracción escondiendo su estructura y secretos de comportamiento.
2. Visión interna (implementación), que abarca el código que se ofrece en la interfaz de la clase.

2.7.1. Relaciones entre clases

Representan tipos de compartición entre clases, o relaciones semánticas.

1. **Asociación.** Indica relaciones de mandato bidireccionales (punteros ocultos en C++). Conlleva dependencia semántica y no establece una dirección de dependencia. Tienen cardinalidad.
2. **Herencia.** Por esta relación una clase (subclase) comparte la estructura o comportamiento definidos en una (herencia simple) o más (herencia múltiple) clases, llamadas superclases:
   — Representa una relación del tipo “es un” entre clases.
   — Una subclase aumenta o restringe el comportamiento o estructura de la superclase (o ambas cosas).
   — Una clase de la que no existen ejemplos se denomina {\it abstracta}.
   — C++ declara como virtuales todas aquellas funciones que quiere modificar en sus subclases.

3. **Agregación.** Representa una relación del tipo "tener un" entre clases. Cuando la clase contenida no existe independientemente de la clase que la contiene se denomina agregación por valor y además implica contenido físico, mientras que si existe independientemente y se accede a ella indirectamente, es agregación por referencia.

4. **Uso.** Es un refinamiento de la asociación donde se especifica cuál es el cliente y cuál el servidor de ciertos servicios, permitiendo a los clientes acceder solo a las interfaces públicas de los servidores, ofreciendo mayor encapsulación de la información.

5. **Ejemplificación.** Se usa en lenguajes que soportan genericidad (declaración de clases parametrizadas y argumentos tipo "template"). Representa las relaciones entre las clases parametrizadas, que admiten parámetros formales, y las clases obtenidas cuando se concretan estos parámetros formales, ejemplificados o inicializados con un ejemplo.

6. **Metaclases.** Son clases cuyos ejemplos son a su vez clases. No se admiten en C++.

2.7.2. Relaciones entre clases y objetos

— Todo objeto es el ejemplo de una clase y toda clase tiene 0 o más objetos.

— Mientras las clases son estáticas, con semántica, relaciones y existencia fijas previamente a la ejecución de un programa, los objetos se crean y destruyen rápidamente durante la actividad de una aplicación.

El diseño de clases y objetos es un proceso incremental e iterativo. Debe asegurar la optimización en los parámetros:

1. **Acoplamiento:** grado de acoplamiento entre módulos.

2. **Cohesión:** mide el grado de conectividad entre elementos de un módulo y entre objetos de una clase.

3. **Suficiencia:** indica que las clases capturan suficientes características de la abstracción para conseguir un comportamiento e interacción eficiente y con sentido.

4. **Completitud:** indica que la interface de la clase captura todo el significado característico de una abstracción, escrito en el mínimo espacio.

5. **Primitividad:** las operaciones deben implementarse si dan acceso a una representación fundamental de la abstracción. Cuáles son operaciones primitivas y cuáles no (se pueden realizar a partir de otras) es un asunto subjetivo y afecta a la eficiencia en la implementación.

2.8. Objeto

La construcción clase (class) de JAVA soporta abstracción y encapsulación de datos. Una clase describe la construcción de un objeto y sirve como plano para construirlo; especifica su funcionamiento interno y su interfaz pública. Cada clase tiene un nombre y especifica a los miembros que pertenecen a ella; estos pueden ser campos (datos) y métodos (funciones). Una vez que se define una clase, su nombre se vuelve un nuevo tipo de dato y se usa para declarar variables de ese tipo y crear objetos de ese tipo.

Una vez que se ha declarado una clase pueden crearse sus objetos. La definición de una clase es el plano de construcción de los objetos, y estos se conocen como instancias de la clase. El nombre de esta se vuelve un nombre de tipo y puede utilizarse para declarar variables. Una variable de tipo de clase es una variable de referencia que puede contener la dirección de memoria (o referencia) de un objeto de la clase o null para una referencia no válida.

La inicialización del objeto creado de una clase se hace por medio del constructor (un método especial de una clase). El operador **new** de JAVA asigna espacio dinámicamente (al momento de la ejecución) y se utiliza para crear objetos. En lenguajes como JAVA, todos los objetos se crean al momento de la ejecución con el operador **new**.

Veamos un ejemplo:

Tenemos una clase llamada **cochazo** y queremos crear un objeto llamado **mi coche** de esa clase:

```
cochazo mi_coche = new cochazo();
```

o

```
cochazo mi_coche;
mi_coche = new cochazo();
```

El objeto es un concepto, una abstracción o una cosa con unos límites definidos y que es relevante para el tema en cuestión, podemos decir además que estos poseen identidad y son distinguibles, aunque dos objetos tengan los mismos valores para todos, sus atributos son diferentes.

En la vida real **se llama objeto a cualquier cosa real o abstracta, en la cual podemos almacenar datos y los métodos para controlar dichos datos.**

— Un objeto es una cosa tangible, algo a que se puede aprender intelectualmente o algo hacia lo que se puede dirigir una acción o pensamiento.

— Un objeto representa un ítem individual e identificable, o una entidad real o abstracta, con un papel definido en el dominio del problema.

— Un objeto tiene:
  - Estado.
  - Comportamiento.
  - Identidad.

La estructura y el comportamiento de objetos similares se definen en sus clases comunes. El término objeto e instancia de una clase son idénticas.

Estado de un objeto. El estado de un objeto abarca todas las propiedades del objeto, y los valores actuales de cada una de esas propiedades. Las propiedades de los objetos suelen ser estáticas, mientras los valores que toman estas propiedades cambian con el tiempo.

— El hecho de que los objetos tengan estado implica que ocupan un espacio, ya en el mundo físico, ya en la memoria del ordenador.

— El estado de un objeto está influido por la historia del objeto.

— No deben confundirse los objetos, que existen en el tiempo, son mutables, tienen estado, pueden ser creados, destruidos y compartidos..., con los valores (los asignados a una variable, por ejemplo) que son cantidades con las propiedades de ser atemporales, inmutables.

— El estado de un objeto representa el efecto acumulado de su comportamiento.

Identidad de un objeto. Identidad es la propiedad de un objeto que lo lleva a distinguirse de otros.

Comportamiento de un objeto. Comportamiento es como un objeto actúa y reacciona, en términos de sus cambios de estado y de los mensajes que intercambia.

El comportamiento de un objeto representa su actividad externamente visible y testable. Son las operaciones que una clase realiza (llamadas también mensajes) las que dan cuenta de cómo se comporta la clase. Por operación se denota el servicio que una clase ofrece a sus clientes. Un objeto puede realizar cinco tipos de **operaciones** sobre otro, con el propósito de provocar una reacción:

1. **Modificador:** altera el estado de un objeto.
2. **Selector:** accede al estado de un objeto, sin alterarlo.
3. **Iterador:** permite a todas las partes de un objeto ser accedidas en un orden.
4. **Constructor:** crea un objeto o inicializa su estado.
5. **Destructor:** libera el estado de un objeto o destruye el objeto.

C++ soporta, además de las operaciones, subprogramas libres. En la terminología de C++ las operaciones que un cliente puede realizar sobre un objeto se declaran como funciones miembro.

2.8.1. Relaciones entre objetos

Las relaciones entre objetos abarcan las operaciones, resultados y suposiciones que unos hacen sobre los otros.

1. **Links:** son conexiones físicas o conceptuales entre objetos. Denota la asociación específica por la que un objeto (cliente) usa o solicita el servicio de otro objeto (servidor). El paso de mensajes entre objetos los sincroniza.

2. **Agregaciones:** denota relaciones todo/parte, con capacidad para gobernar desde el todo las partes. Es equivalente a la relación "tener un". El todo puede contener a la parte.

Agregación es conveniente en las ocasiones en que el encapsulamiento de las partes es prioritario. Si se requiere que las relaciones entre objetos estén vagamente acopladas, se utilizan links.

2.9. Propiedades y métodos

El método es la especificación de un proceso de una operación, es un proceso disciplinado para generar un conjunto de modelo que describen varios aspectos de un sistema de software en desarrollo, utilizando alguna notación bien definida.

Los métodos especifican la forma en que se controlan los datos de un objeto. Los métodos en un tipo de objeto solo hacen referencia a las estructuras de datos de ese tipo de objeto. No deben tener acceso directo a las estructuras de datos de otros objetos. Para utilizar la estructura de datos de otro objeto debe enviar un mensaje a este. El tipo de objeto empaca juntos los tipos de datos y los métodos.

2.9.1. Propiedades

Todo objeto puede tener cierto número de propiedades, cada una de las cuales tendrá, a su vez, uno o varios valores. En POO, las propiedades corresponden a las clásicas variables de la programación estructurada. Son, por lo tanto, datos encapsulados dentro del objeto, junto con los métodos (programas) y las relaciones (punteros a otros objetos). Las propiedades de un objeto pueden tener un valor único o pueden contener un conjunto de valores más o menos estructurados (matrices, vectores, listas, etc.). Además, los valores pueden ser de cualquier tipo (numérico, alfabético, etc.) si el sistema de programación lo permite.

Pero existe una diferencia con las variables, y es que las propiedades se pueden heredar de unos objetos a otros. En consecuencia, un objeto puede tener una propiedad de maneras diferentes:

— **Propiedades propias.** Están formadas dentro de la cápsula del objeto.

— **Propiedades heredadas.** Están definidas en un objeto antepasado del actual. A veces estas propiedades se llaman propiedades miembro porque el objeto las posee por el mero hecho de ser miembro de una clase.

2.9.2. Métodos

Podemos definir al método como un programa procedimental o procedural escrito en cualquier lenguaje, que está asociado a un objeto determinado y cuya ejecución solo puede desencadenarse a través de un mensaje recibido por este o por sus descendientes.

Son sinónimos de método todos aquellos términos que se han aplicado tradicionalmente a los programas, como procedimiento, función, rutina, etc. Sin embargo, es conveniente utilizar el término método para que se distingan claramente las propiedades especiales que adquiere un programa en el entorno POO, que afectan fundamentalmente a la forma de invocarlo (únicamente a través de un mensaje) y a su campo de acción, limitado a un objeto y a sus descendientes (no tiene por qué a todos).

Dado que los métodos son partes de los programas pueden tener argumentos o parámetros. Puesto que los métodos pueden heredarse de unos objetos a otros, un objeto puede disponer de un método de dos maneras diferentes (al igual que sucedía con las propiedades):

— **Métodos propios.** Están incluidos dentro de la cápsula del objeto.

— **Métodos heredados.** Están definidos en un objeto antepasado del actual. A veces estos métodos se llaman métodos miembro porque el objeto los posee por el mero hecho de ser miembro de una clase.

— La **sobrecarga de métodos** consiste en poner varios métodos con el mismo nombre en la misma clase, pero siempre que su lista de argumentos sea distinta. El compilador sabría a cuál de todas las sobrecargas nos referimos por los argumentos que se le pasen en la llamada. Lo que diferencia las listas de argumentos de las diferentes sobrecargas no es el nombre de las variables, sino el tipo de cada una de ellas.

  Los métodos sobrecargados dan a los programadores la flexibilidad de llamar a un método similar para diferentes tipos de datos.

Un método es un procedimiento de cálculo definido en una clase. Cada método contiene instrucciones que especifican una secuencia de acciones de cálculo que habrán de realizarse, y variables que se utilizan para almacenar y producir los valores necesarios. Algunas de las variables pueden ser objetos y el proceso de cálculo incluye interacciones entre los mismos, generalmente.

Un método toma argumentos como entrada, realiza una secuencia de pasos programados y devuelve un resultado del tipo declarado (existen métodos que no devuelven obligatoriamente un valor, pueden sencillamente cambiar el valor de una propiedad o inicializar una de esas propiedades). También puede llamar a otros métodos en el curso de sus cálculos.

Una definición de método contiene un encabezado y un cuerpo. El encabezado define el nombre del método y el tipo del valor de regreso. El encabezado también especifica variables, conocidas como parámetros formales, que reciben los argumentos de entrada y se utilizan en el cuerpo del método para realizar cálculos.

Mientras que el cuerpo incluye una secuencia de declaraciones e instrucciones encerradas entre llaves, { }, una declaración proporciona información al compilador y una instrucción especifica las acciones que habrán de ejecutarse.

En general, a las propiedades y a los métodos de una clase se les suele llamar miembros de datos pues realmente son los únicos elementos que aparecen físicamente dentro de la clase. Antes de pretender crear un molde para hacer objetos (clase) nos debemos plantear muy a conciencia qué propiedades y qué métodos queremos que tenga esa clase.

Hoy en día casi todos los lenguajes de programación se basan en propiedades y métodos, pudiendo el usuario del programa cambiar los valores de estas propiedades a su gusto (ejemplo: escritorio, propiedad: papel tapiz, podemos nosotros configurarlo a nuestro antojo).

2.10. Mensajes

Para que el objeto haga algo, enviamos una solicitud. Esta hace que se produzca una operación. La operación ejecuta el método apropiado y, de manera opcional, produce una respuesta. El mensaje que constituye la solicitud contiene el nombre del objeto, el nombre de una operación, a veces, un grupo de parámetros.

Un mensaje es una solicitud para que se lleve a cabo la operación indicada y se produzca el resultado. Por tanto, los mensajes son solicitudes que invocan operaciones específicas, con uno o más objetos como parámetros. La respuesta a estas órdenes será la información requerida, siempre que el objeto considere que quien envía el mensaje está autorizado para obtenerla.

2.11. Identidad

La identidad es aquella propiedad de un objeto que los distingue de todos los demás. La identidad única (pero no necesariamente el nombre) de cada objeto se preserva durante el tiempo de vida del mismo, incluso cuando su estado cambia. La identidad es la naturaleza de un objeto que lo distingue de todos los demás.

2.12. Reutilización o reusabilidad

Es volver a generar una clase, teniendo en cuenta que puede ser útil para varios sistemas, sin tener que volver a generarlos, ahorrando con esto tiempo para programación, etc. **Las clases están definidas para que se reutilicen en muchos sistemas**. Para que esta sea efectiva, las clases se deben construir a partir de un modo, que puedan ser adaptables y reutilizables indefinidamente.

Un objetivo de las técnicas orientadas a objetos es lograr la reutilización masiva al construir un software. Los sistemas suelen ser construidos a través de objetos ya existentes, que se lleva a un alto grado de reutilización, esto conlleva a un ahorro de dinero, un menor tiempo de desarrollo y una mayor confiabilidad de sistemas.

Por lo tanto, si ya hemos puesto a prueba una clase en un sistema, tendremos la garantía y la confiabilidad que podrá volver a ser reutilizada.

2.13. Jerarquía

La jerarquía es una clasificación u ordenación de abstracciones. Las dos jerarquías más importantes en un sistema complejo son su **estructura de clases y su estructura de objetos**, jerarquía de clase y jerarquía de partes respectivamente.

Como ya habíamos mencionado anteriormente la herencia es el ejemplo más claro de una jerarquía de clases. Esta define una relación entre clases, en la que una clase comparte la estructura de comportamiento definida en una o más clases (herencia simple o herencia múltiple, respectivamente).

2.14. Concurrencia, persistencia y tipificado

2.14.1. Concurrencia

Es la propiedad que distingue un objeto activo de uno no activo. Concurrencia permite que diferentes objetos actúen al mismo tiempo, usando distintos threads (hilos) de control.

Para cierto tipos de problemas, un sistema automatizado puede tener que manejar muchos eventos diferentes simultáneamente, en otro problema pueden implicar tantos cálculos que excedan la capacidad de cualquier procesador individual. En ambos casos es natural considerar el uso de un conjunto distribuido de computadores para la implantación que se persigue o utilizar procesadores capaces de realizar multitareas, a través de un hilo de control, mediante el cual se producen acciones dinámicas independientes dentro del sistema. **La concurrencia permite a diferentes objetos actuar al mismo tiempo.**

2.14.2. Persistencia

Es la propiedad por la cual la existencia de un objeto trasciende en el tiempo (el objeto sigue existiendo después de que su creador deja de existir) o en el espacio (la localización del objeto cambia respecto a la dirección en la que fue creado).

2.14.3. Tipificado

Tipificar es la imposición de una clase a un objeto, de tal modo que objetos de diferentes tipos no se puedan intercambiar o se puedan intercambiar solo de forma restringida.

— Tipo es una caracterización precisa de las propiedades estructurales y de comportamiento que comparten una colección de entidades.

— Una clase define un nuevo tipo de objetos, por tanto, clase y tipo muchos programadores los consideran sinónimos.

— Existen lenguajes fuertemente tipificados (Ada) y débilmente tipificados. Estos últimos soportan polimorfismo, mientras que los fuertemente tipificados no.

2.15. Modularidad

La modularidad es la propiedad que posee un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados. La modularización consiste en dividir un programa en módulos que pueden compilarse separadamente, pero que tiene conexiones con otros módulos. Así, los principios de abstracción, encapsulamiento y modularidad son sinérgicos (puede haber muchas más abstracciones diferentes de las que se pueden comprender simultáneamente, el encapsulamiento ayuda a manejar esta complejidad ocultando la visión interna de las abstracciones. La modularidad ayuda ofreciendo una vía para agrupar abstracciones relacionadas lógicamente).

2.16. Relaciones entre los conceptos asociados al modelo de objetos

— Los conceptos de abstracción y encapsulación son conceptos complementarios: abstracción hace referencia al comportamiento observable de un objeto, mientras encapsulación hace referencia a la implementación que la hace alcanzar este comportamiento.

— Existe una tensión entre los conceptos de encapsulación de la información y el concepto de jerarquía de herencia, que requiere una apertura en el acceso a la información.

— C++ ofrece mucha flexibilidad, pudiendo disponer de tres compartimentos en cada clase:

1. **Privado**: declaraciones accesibles solo a la clase (completamente encapsulado).
2. **Protegido**: declaraciones accesibles a la clase y a sus subclases.
3. **Público**: declaraciones accesibles a todos los clientes.

Además de estos tres tipos, C++ soporta la definición de clases cooperativas a las que se les permite acceder a la parte privada de la implementación. Estas clases se denominan **friends**.

Vamos a realizar un esquema de cada uno de los términos vistos:

OBJETO	Fin, intento, propósito. Materia y sujeto de una ciencia.
CLASE	Orden de cosas de una misma especie. Conjunto de órdenes.
POLIMORFISMO	Propiedad de los cuerpos que cambian de forma sin cambiar su naturaleza. Presencia de distintas formas individuales en una sola especie.
ENCAPSULACIÓN	Proceso de constitución de una cápsula.
HERENCIA	Derecho de suceder a otro la posesión de bienes o acciones.
MENSAJE	Información que se le envía a alguien.
MÉTODO	Modo de hacer en orden una cosa, modo habitual de proceder.
IDENTIDAD	Cualidad de ser lo mismo que otra cosa con que se compara.
REUSABILIDAD	Acción de volver a utilizar. Que puede volver a ser utilizado.
JERARQUÍA	Orden o grados de una especie.
ABSTRACCIÓN	Separación, apartamiento, aislamiento, prescindir.
CONCURRENCIA	Asistencia, reunión simultánea de personas o cosas.
MODULARIDAD	Acción de pasar de un término a otro.

3. Ventajas e inconvenientes

3.1. Ventajas de la POO

El costo del diseño, la implantación, la verificación, el mantenimiento y la revisión de sistemas grandes de software es muy alto. Por tanto, es importante encontrar maneras de facilitar estas tareas. En este sentido, POO tiene un potencial enorme.

Los sistemas orientados a objetos son también más resistentes al cambio y, por lo tanto, están mejor preparados para evolucionar en el tiempo, porque su diseño esta basado en formas intermedias estables.

El modelo de objetos ha influido incluso en las fases iniciales del ciclo de vida del desarrollo del software. El análisis orientado a objetos (AOO) enfatiza la construcción de modelos del mundo real utilizando una visión del mundo orientado a objetos. Es un método de análisis que examina los requisitos desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema.

Básicamente los productos del análisis orientado a objetos sirven como modelos de los que se puede partir para un diseño orientado a objetos; los productos del diseño orientado a objetos pueden utilizarse entonces como anteproyectos para la implementación completa de unos sistemas utilizando métodos de programación orientada a objetos, de esta forma se relacionan AOO, DOO y POO.

La programación orientada a objetos ofrece las siguientes ventajas principales:

— **Simplicidad:** como los objetos de software son modelos de objetos reales en el dominio de la aplicación, la complejidad del programa se reduce y su estructura se vuelve clara y simple.

— **Modularidad:** cada objeto forma una entidad separada cuyo funcionamiento interno está desacoplado de otras partes del sistema.

— **Facilidad para hacer modificaciones:** es sencillo hacer cambios menores en la representación de los datos o los procedimientos utilizados en un programa Orientado a Objetos (O.O.). Las modificaciones hechas en el interior de un objeto no afectan ninguna otra parte del programa, siempre y cuando se preserve su comportamiento externo.

— **Posibilidad de extenderlo:** la adición de nuevas funciones o la respuesta a ambientes operativos cambiantes puede lograrse con solo introducir algunos objetos nuevos y variar algunos existentes.

— **Flexibilidad:** un programa Orientado a Objetos (O.O.) puede ser muy manejable al adaptarse a diferentes situaciones, porque es posible cambiar los patrones de interacción entre los objetos sin alterarlos.

— **Facilidad para darle mantenimiento:** los objetos pueden mantenerse por separado, lo que facilita la localización y el arreglo de problemas, así como la adición de otros elementos.

— **Reusabilidad:** los objetos pueden emplearse en diferentes programas. Por ejemplo, si se tiene uno que construye tablas puede utilizarse en cualquier programa que requiera cierto tipo de tabla. Por tanto, es posible construir programas a partir de componentes prefabricados y preprobados en una fracción del tiempo requerido para elaborar nuevos programas desde el principio.

— **Dominio del problema:** el paradigma O.O. es más que una forma de programar. Es una forma de pensar acerca de un problema desde el punto de vista del mundo real en vez de desde el punto de vista del ordenador. El AOO permite analizar mejor el dominio del problema, sin pensar en términos de implementar el sistema de un ordenador, permite, además, pasar directamente el dominio del problema al modelo del sistema.

— **Comunicación:** el concepto O.O. es más simple y está menos relacionado con la informática que el concepto de flujo de datos. Esto permite una mejor comunicación entre el analista y el experto en el dominio del problema.

— **Consistencia:** los objetos encapsulan tanto atributos como operaciones. Debido a esto, el AOO reduce la distancia entre el punto de vista de los datos y el punto de vista del proceso, dejando menos lugar a inconsistencias y disparidades entre ambos modelos.

— **Expresión de características comunes:** el paradigma lo utiliza la herencia para expresar explícitamente las características comunes de una serie de objetos; estas característica comunes quedan escondidas en otros enfoques y llevan a duplicar entidades en el análisis y código en los programas. Sin embargo, el paradigma O.O. pone especial énfasis en la reutilización y proporciona mecanismos efectivos que permiten reutilizar aquello que es común sin impedir por ello describir las diferencias.

— **Resistencia al cambio:** los cambios en los requisitos afectan notablemente a la funcionalidad de un sistema por lo que afectan mucho al software desarrollando con métodos estructurados. Sin embargo, los cambios afectan en mucha menos medida a los objetos que componen o maneja el sistema, que son mucho más estables. Las modificaciones necesarias para adaptar una aplicación basada en objetos a un cambio de requisitos suelen estar mucho más localizadas.

— **Reutilización:** aparte de la reutilización interna, basada en la expresión explícita de características comunes, el paradigma O.O. desarrolla modelos mucho más próximos al mundo real, con lo que aumentan las posibilidades de reutilización. Es probable que en futuras aplicaciones nos encontremos con objetos iguales o similares a los de la actual.

Todos los problemas aún no han sido solucionados en forma completa. Pero como los objetos son portables, mientras que la herencia permite la reusabilidad del código orientado a objetos, es más sencillo modificar código existente porque los objetos no interaccionan excepto a través de mensajes; en consecuencia, un cambio en la codificación de un objeto no afectará la operación con otro objeto siempre que los métodos respectivos permanezcan intactos. La introducción de tecnología de objetos como una herramienta conceptual para analizar, diseñar e implementar aplicaciones permite obtener aplicaciones más modificables, fácilmente extensibles y a partir de componentes reusables. Esta reusabilidad del código disminuye el tiempo que se utiliza en el desarrollo y hace que el desarrollo del software sea más intuitivo porque la gente piensa naturalmente en términos de objetos más que en términos de algoritmos de software.

3.2. Inconvenientes de la POO

En un sistema orientado a objetos los problemas (si los hubiese) suelen surgir en la implementación de tal sistema. Muchas compañías oyen acerca de los beneficios de un sistema orientado a objetos e invierten gran cantidad de recursos, luego comienzan a darse cuenta de que han impuesto una nueva cultura que es ajena a los programadores actuales. A pesar de ser muy pocos los inconvenientes de la POO, citaremos los siguientes:

— **Curvas de aprendizaje largas.** Al hacer la transición a un sistema orientado a objetos la mayoría de los programadores deben formarse nuevamente antes de poder usarlo.

— **Dependencia del lenguaje.** La elección de un lenguaje orientado a objetos u otro tiene ramificaciones de diseño muy importantes.

— **Determinación de las clases.** Una clase es un molde que se utiliza para crear nuevos objetos. Si bien hay muchas jerarquías de clase predefinidas usualmente se deben crear clases específicas para la aplicación que se esté desarrollando. En consecuencia, es importante crear el conjunto de clases adecuado para un proyecto.

— **Descomposición funcional.** El análisis estructurado se basa en la descomposición funcional del sistema que queremos construir. El problema es que no existe un mecanismo para comprobar si la especificación del sistema expresa con exactitud los requisitos del mismo.

— **Flujos de datos.** El análisis estructurado muestra cómo fluye la información a través del sistema. Aunque este enfoque se adapta bien al uso de sistemas informáticos para implementar al sistema, no es la forma habitual de pensar.

— **Modelo de datos.** El análisis estructurado moderno incorpora modelos de datos, además de modelos de procesos y de comportamiento. Sin embargo, la relación entre los modelos es muy débil y hay muy poca influencia de un modelo en otro.

4. Patrones de diseño y lenguaje de modelado unificado (UML)

4.1. UML

UML (Lenguaje Unificado de Modelado -*Unified Modeling Language*-), es el lenguaje de modelado de sistemas de software. Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software. UML ofrece un estándar para describir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como procesos de negocios y funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes de software reutilizables.

Es importante remarcar que UML es un lenguaje para especificar y no un método o un proceso, se utiliza para definir un sistema de software, para detallar los artefactos en el sistema y para documentar y construir –es el lenguaje en el que está descrito el modelo–. Se puede aplicar en una gran variedad de formas para soportar una metodología de desarrollo de software pero no especifica en sí mismo qué metodología o proceso usar.

UML es un lenguaje. Un lenguaje proporciona un vocabulario y unas reglas para permitir una comunicación. Este lenguaje nos indica cómo crear y leer los modelos, pero no dice cómo crearlos. Esto último es el objetivo de las metodologías de desarrollo. Los objetivos de UML son:

— Visualizar, expresa de forma gráfica.

— Especificar las características de un sistema.

— Construir a partir de los modelos especificados.

— Documentar, los propios elementos gráficos sirven de documentación.

4.1.1. Bloques de construcción de un modelo UML

Un modelo UML está compuesto por tres clases de bloques de construcción:

A) Elementos

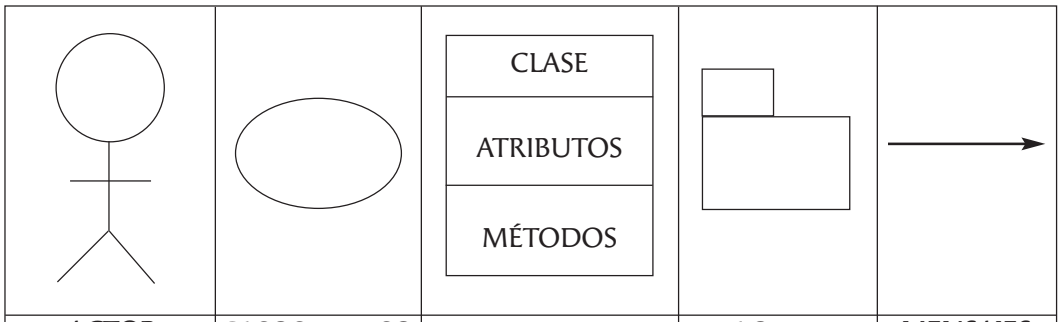
Los elementos son abstracciones de cosas reales o ficticias (objetos, acciones, etc.).

Pueden ser estructurales, de comportamiento de agrupación o de anotación.

— **Elementos estructurales**: actores, casos de uso, clases, objetos.

— **Elementos de comportamiento**: mensajes.

— **Elementos de agrupación**: paquetes.



ACTOR	CASOS DE USO		PAQUETE	MENSAJES

B) Relaciones

Relacionan los elementos entre sí. Las relaciones pueden ser del tipo:

- **Dependencia**

Es una relación semántica entre dos elementos, en la cual un cambio en un elemento puede afectar a la semántica de otro elemento. Existen varios tipos de dependencia predefinidas que se indican mediante extend o include para casos de uso.

- **Asociación**

Es una relación estructural entre dos elementos, que describen las conexiones entre ellos (suele ser bidireccional). Puede presentarse como agregación o composición.

- **Generalización**

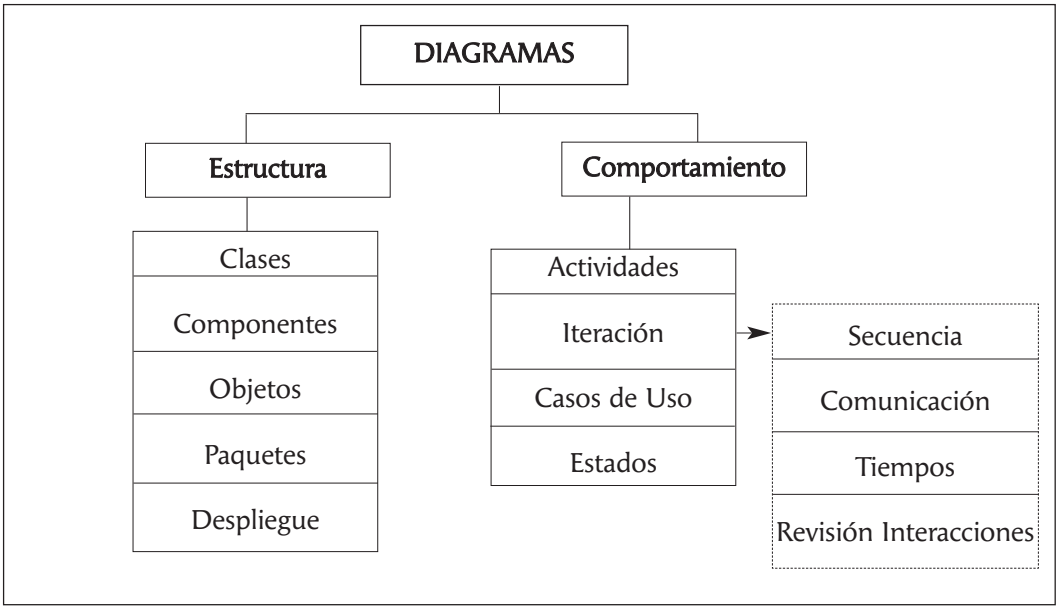
Es una relación entre un elemento más general (el padre) y un elemento más específico (el hijo).

- **Implementación**

Es una relación en la que un elemento (hijo) realiza las acciones indicadas por el padre.

4.2. Diagramas

Son colecciones de elementos con sus relaciones. UML cuenta con varios tipos de diagramas, los cuales muestran diferentes aspectos de las entidades representadas. Los veremos en esta sección:



4.2.1. Diagramas de estructura

Enfatizan en los elementos que deben existir en el sistema modelado. Los diagramas estructurales representan elementos y así componen un sistema o una función. Estos diagramas pueden reflejar las relaciones estáticas de una estructura, como lo hacen los diagramas de clases o de paquetes, o arquitecturas en tiempo de ejecución, tales como diagramas de objetos o de estructura de composición.

- **Diagrama de clases**

Los diagramas de clases son utilizados durante el proceso de análisis y diseño de los sistemas informáticos, donde se crea el diseño conceptual de la información que se manejará en el sistema, los componentes que se encarga-

ran del funcionamiento y la relación entre uno y otro. Muestra una colección de elementos de modelado declarativo (estáticos), tales como clases, tipos y sus contenidos y relaciones. Se basan en:

— **Propiedades:** también llamados atributos o características, son valores que corresponden a un objeto, como color, material, cantidad, ubicación. Generalmente se conoce como la información detallada del objeto. Suponiendo que el objeto es una puerta, sus propiedades serían: marca, tamaño, color y peso.

— **Operaciones:** son aquellas actividades o verbos que se pueden realizar con/para este objeto, como por ejemplo abrir, cerrar, buscar, cancelar, acreditar, cargar. De la misma manera que el nombre de un atributo, el nombre de una operación se escribe con minúsculas si consta de una sola palabra. Si el nombre contiene más de una palabra, cada palabra será unida a la anterior y comenzará con una letra mayúscula, a excepción de la primera palabra que comenzará en minúscula. Por ejemplo: abrirPuerta, cerrarPuerta, buscarPuerta, etc.

— **Interfaz:** es un conjunto de operaciones o propiedades que permiten a un objeto comportarse de cierta manera, por lo que define los requerimientos mínimos del objeto.

— **Herencia:** se define como la reutilización de un objeto padre ya definido para poder extender la funcionalidad en un objeto hijo. Los objetos hijos heredan todas las operaciones o propiedades de un objeto padre. Por ejemplo: una persona puede subdividirse en Proveedores, Acreedores, Clientes, Accionistas, Empleados; todos comparten datos basicos como una persona, pero además tendrá información adicional que depende del tipo de persona, como saldo del cliente, total de inversión del accionista, salario del empleado, etc.

- **Diagrama de componentes**

Un diagrama de componentes representa la separación de un sistema de "sofware" en componentes físicos (por ejemplo archivos, cabeceras, módulos, paquetes, etc.) y muestra las dependecias entre estos componentes. Representa los componentes que componen una aplicación, sistema o empresa. Los componentes, sus relaciones, interacciones y sus interfaces públicas.

Los diagramas de componentes describen los elementos físicos del sistema y sus relaciones. Muestran las opciones de realización incluyendo código fuente, binario y ejecutable. Los componentes representan todos los tipos de elementos software que entran en la fabricación de aplicaciones informáticas. Pueden ser simples archivos, paquetes de Ada, bibliotecas cargadas dinámicamente, etc. Las relaciones de dependencia se utilizan en los diagramas de componentes para indicar que un componente utiliza los servicios ofrecidos por otro componente.

Un diagrama de componentes representa las dependencias entre componentes software, incluyendo componentes de código fuente, componentes del código binario y componentes ejecutables. Un módulo de software se puede

representar como componente. Algunos componentes existen en tiempo de compilación, algunos en tiempo de enlace y algunos en tiempo de ejecución, otros en varias de estas.

Un componente de solo compilación es aquel que es significativo únicamente en tiempo de compilación. Un componente ejecutable es un programa ejecutable.

Un diagrama de componentes tiene solo una versión con descriptores, no tiene versión con instancias. Para mostrar las instancias de los componentes se debe usar un diagrama de despliegue.

Un diagrama de componentes muestra clasificadores de componentes, las clases definidas en ellos y las relaciones entre ellas. Los clasificadores de componentes también se pueden anidar dentro de otros clasificadores de componentes para mostrar relaciones de definición.

Un diagrama que contiene clasificadores de componentes y de nodo se puede utilizar para mostrar las dependencias del compilador, que se representa como flechas con líneas discontinuas (dependencias) de un componente cliente a un componente proveedor del que depende. Los tipos de dependencias son específicos del lenguaje y se pueden representar como estereotipos de las dependencias.

El diagrama también puede usarse para mostrar interfaces y las dependencias de llamada entre componentes, usando flechas con líneas discontinuas desde los componentes a las interfaces de otros componentes.

El diagrama de componente hace parte de la vista física de un sistema, la cual modela la estructura de implementación de la aplicación por sí misma, su organización en componentes y su despliegue en nodos de ejecución. Esta vista proporciona la oportunidad de establecer correspondecias entre las clases y los componentes de implementación y nodos. La vista de implementación se representa con los diagramas de componentes.

- **Diagrama de objetos**

Los diagramas de objetos son utilizados durante el proceso de análisis y diseño de los sistemas informáticos en la metodología UML.

Se puede considerar un caso especial de un diagrama de clases en el que se muestran instancias específicas de clases (objetos) en un momento particular del sistema. Los diagramas de objetos utilizan un subconjunto de los elementos de un diagrama de clase. Los diagramas de objetos no muestran la multiplicidad ni los roles, aunque su notación es similar a los diagramas de clase. Una diferencia con los diagramas de clase es que el compartimiento de arriba va en la forma, Nombre de objeto: Nombre de clase. Por ejemplo, Miguel: Persona.

Un diagrama que presenta los objetos y sus relaciones en un punto del tiempo. Un diagrama de objetos se puede considerar como un caso especial de un diagrama de clases o un diagrama de comunicaciones.

- **Diagrama de despliegue**

El diagrama de despliegue es un tipo de diagrama del lenguaje unificado de modelado que se utiliza para modelar el hardware utilizado en la implementaciones de sistemas y las relaciones entre sus componentes.

Los elementos usados por este tipo de diagrama son nodos (representados como un prisma), componentes (representados como una caja rectangular con dos protuberancias del lado izquierdo) y asociaciones.

En el UML 2.0 los componentes ya no están dentro de nodos. En cambio, puede haber artefactos u otros nodos dentro de un nodo.

Un artefacto puede ser algo como un archivo, un programa, una biblioteca o una base de datos construida o modificada en un proyecto. Estos artefactos implementan colecciones de componentes. Los nodos internos indican ambientes, un concepto más amplio que el hardware propiamente dicho, ya que un ambiente puede incluir al lenguaje de programación, a un sistema operativo, un ordenador o un cluster de terminales.

Un diagrama de despliegue físico muestra cómo y dónde se desplegará el sistema. Las máquinas físicas y los procesadores se representan como nodos y la construcción interna puede ser representada por nodos o artefactos embebidos. Como los artefactos se ubican en los nodos para modelar el despliegue del sistema, la ubicación es guiada por el uso de las especificaciones de despliegue.

- **Diagrama de paquetes**

En el lenguaje unificado de modelado, un diagrama de paquetes muestra cómo un sistema está dividido en agrupaciones lógicas mostrando las dependencias entre esas agrupaciones. Dado que normalmente un paquete está pensado como un directorio, los diagramas de paquetes suministran una descomposición de la jerarquía lógica de un sistema.

Los paquetes están normalmente organizados para maximizar la coherencia interna dentro de cada paquete y minimizar el acoplamiento externo entre los paquetes. Con estas líneas maestras sobre la mesa, los paquetes son buenos elementos de gestión. Cada paquete puede asignarse a un individuo o a un equipo, y las dependencias entre ellos pueden indicar el orden de desarrollo requerido.

Un diagrama que presenta cómo se organizan los elementos de modelado en paquetes y las dependencias entre ellos, incluyendo importaciones y extensiones de paquetes.

- **Diagrama de estructura de composición**

Representa la estructura interna de un clasificador (una clase, un componente o un caso de uso), incluyendo los puntos de interacción de clasificador con otras partes del sistema.

Los diagramas de composición de estructuras fueron específicamente diseñados para la representación de patrones de diseño y son una de las modificaciones de mayor impacto dentro de UML 2.0. Los diagramas de composición de estructuras permiten, potencialmente, documentar arquitecturas de software de manera un poco más clara que en versiones anteriores del UML 2.0.

4.2.2. Diagramas de comportamiento

Enfatizan en lo que debe suceder en el sistema modelado. Los diagramas de comportamiento representan las características de comportamiento de un sistema o proceso de negocios y, a su vez, incluyen a los diagramas de: actividades, casos de uso, máquinas de estados, tiempos, secuencias, repaso de interacciones y comunicaciones.

- **Diagrama de actividades**

En el lenguaje de modelado unificado, un diagrama de actividades representa los flujos de trabajo paso a paso de negocio y operacionales de los componentes en un sistema. Un diagrama de actividades muestra el flujo de control general.

En SysML el diagrama de actividades ha sido extendido para indicar flujos entre pasos que mueven elementos físicos (gasolina) o energía (presión). Los cambios adicionales permiten al diagrama soportar mejor flujos de comportamiento y datos continuos.

Muchos cambios fueron realizados en los diagramas de actividad en la versión 2 de UML. Los cambios realizados son tendentes a:

— Dar soporte en la definición de procesos de negocio.

— Brindar una semántica similar al de las redes de Petri.

— Permitir una mayor y más flexible representación de paralelismo.

Representa los procesos de negocios de alto nivel, incluidos el flujo de datos. También puede utilizarse para modelar lógica compleja o paralela dentro de un sistema.

- **Diagrama de casos de uso**

En el lenguaje de modelado unificado, un diagrama de casos de uso es una especie de diagrama de comportamiento. El lenguaje de modelado unificado define una notación gráfica para representar casos de uso llamada modelo de casos de uso. UML no define estándares para que el formato escrito describa los casos de uso, y así mucha gente no entiende que esta notación gráfica define la naturaleza de un caso de uso; sin embargo una notación gráfica puede solo dar una vista general simple de un caso de uso o un conjunto de casos de uso. Los diagramas de casos de uso son a menudo confundidos con los casos de uso. Mientras los dos conceptos están relacionados, los casos de uso son mucho más detallados que los diagramas de casos de uso.

El estándar de lenguaje de modelado unificado de OMG define una notación gráfica para realizar diagramas de casos de uso, pero no el formato para describir casos de uso. Mucha gente sufre la equivocación pensando que un caso de uso es una notación gráfica (o es su descripción). Mientras la notación gráfica y las descripciones son importantes, son documentación de un caso de uso: un propósito para el que el actor puede usar el sistema. Los diagramas de casos de uso son diagramas que muestran las relaciones entre actores y el sistema.

Un diagrama muestra las relaciones entre los actores y el sujeto (sistema), y los casos de uso.

El valor verdadero de un caso de uso reposa en dos áreas:

— La descripción escrita del comportamiento del sistema al afrontar una tarea de negocio o un requisito de negocio. Esta descripción se enfoca en el valor suministrado por el sistema a entidades externas tales como usuarios humanos u otros sistemas.

— La posición o contexto del caso de uso entre otros casos de uso. Dado que es un mecanismo de organización, un conjunto de casos de uso coherentes, consistentes promueve una imágen fácil del comportamiento del sistema, un entendimiento común entre el cliente/propietario/usuario y el equipo de desarrollo.

Es práctica común crear especificaciones suplementarias para capturar detalles de requisitos que caen fuera del ámbito de las descripciones de los casos de uso. Ejemplos de esos temas incluyen rendimiento, temas de escalabilidad/gestión o cumplimiento de estándares.

Relaciones de casos de uso. Las tres relaciones principales entre los casos de uso son soportadas por el estándar UML, el cual describe notación gráfica para esas relaciones.

Include. En una forma de interacción, un caso de uso dado puede "incluir" otro. El primer caso de uso a menudo depende del resultado del caso de uso incluido. Esto es útil para extraer comportamientos verdaderamente comunes desde múltiples casos de uso a una descripción individual. La notación es una flecha rayada desde el caso de uso que lo incluye hasta el incluido, con la etiqueta "include". Este uso se asemeja a una expansión de una macro donde el comportamiento del caso incluido es colocado dentro del comportamiento del caso de uso base. No hay parámetros o valores de retorno.

Extend. En otra forma de interacción, un caso de uso dado (la extensión) puede extender a otro. Esta relación indica que el comportamiento del caso de uso extensión puede ser insertado en el caso de uso extendido bajo ciertas condiciones. La notación es una flecha rayada desde el caso de uso extensión al caso de uso extendido, con la etiqueta "extend". Esto puede ser útil para lidiar con casos especiales o para acomodar nuevos requisitos durante el mantenimiento del sistema y su extensión.

Generalization. En la tercera forma de relación entre casos de uso, existe una relación generalización/especialización. Un caso de uso dado puede estar

en una forma especializada de un caso de uso existente. La notación es una línea solida terminada en un triángulo dibujado desde el caso de uso especializado al caso de uso general. Esto se asemeja al concepto orientado a objetos de subclases, en la práctica puede ser útil factorizar comportamientos comunes, restricciones al caso de uso general, se describen una vez, y se trabajan los detalles excepcionales en los casos de uso especializados.

- **Diagrama de estados**

El diagrama de estados de UML es un diagrama de estados con notación estandarizada que puede describir los elementos, desde un programa de computador a procesos de negocio.

Un diagrama de máquina de estados ilustra cómo un elemento, muchas veces una clase, se puede mover entre estados que clasifican su comportamiento, de acuerdo con disparadores de transiciones, guardias de restricciones y otros aspectos de los diagramas de máquinas de estados, que representan y explican el movimiento y el comportamiento.

Lo siguiente son los elementos básicos de notación que pueden usarse para componer un diagrama:

— Círculo lleno, apuntando a un estado inicial.

— Círculo hueco que contiene un círculo lleno más pequeño en el interior, indicando el estado final (si existiera).

— Rectángulo redondeado, denotando un estado. En la parte superior del rectángulo está el nombre del estado. Puede contener una línea horizontal en la mitad, debajo de la cual se indican las actividades que se hacen en el estado.

— Flecha, denotando transición. El nombre del evento (si existiera) que causa esta transición etiqueta el cuerpo de la flecha. Se puede añadir una expresión de Guarda, encerrada en corchetes ([]) denotando que esta expresión debe ser cierta para que la transición tenga lugar. Si se realiza una acción durante la transición, se añade a la etiqueta después de "/". NombreDeEvento[ExpresiónGuarda]/acción.

— Línea horizontal gruesa con x>1 líneas entrando y 1 línea saliendo o 1 línea entrando y x>1 líneas saliendo. Estas denotan Unión/Separación, respectivamente.

Al igual que los diagramas de secuencia, las máquinas de estados permiten una mejor reutilización, a través del agregado de Puntos de Entrada y Puntos de Salida (Entry/Exit Points). Las máquinas de estados son ahora generalizables y soportan una vista centrada en la transición. Las capacidades de generalización incluyen: agregar estados y transiciones, extender estados, reemplazar transiciones, reemplazar maquinas compuestas, etc. Lo que permite que, por ejemplo, dada una clase que hereda de otra, especificar ambas clases mediante máquinas de estados que heredan funcionalidad.

4.2.3. Diagramas de interacción

Son un subtipo de diagramas de comportamiento, que enfatizan sobre el flujo de control y de datos entre los elementos del sistema modelado. El UML 2.0 se encuentra diseñado de manera orientada a objetos, dentro de la nueva organización interna, y cuenta con los llamados "diagramas de interacciones", que son una subcategoría de los diagramas de comportamiento. Estos diagramas muestran la interacción entre distintos clasificadores de un modelo desde distintos puntos de vista, es decir, haciendo foco en distintos aspectos de la interacción. Esto hace que todos los diagramas de interacción tengan ciertas características compartidas, como por ejemplo la capacidad de crear Diagramas de descripción de interacción y la utilización de fragmentos combinados. Dichos conceptos serán descriptos a continuación utilizando los diagramas de secuencias.

- **Diagrama de secuencia**

El diagrama de secuencia es uno de los diagramas más efectivos para modelar interacción entre objetos en un sistema. Un diagrama de secuencia muestra la interacción de un conjunto de objetos en una aplicación a través del tiempo y se modela para cada caso de uso. Mientras que el diagrama de casos de uso permite el modelado de una vista "business" del escenario, el diagrama de secuencia contiene detalles de implementación del escenario, incluyendo los objetos y clases que se usan para implementar el escenario y mensajes pasados entre los objetos. Típicamente uno examina la descripción de un caso de uso para determinar qué objetos son necesarios para la implementación del escenario. Si tiene modelada la descripción de cada caso de uso como una secuencia de varios pasos, entonces puede "caminar sobre" esos pasos para descubrir qué objetos son necesarios para que se puedan seguir los pasos. Un diagrama de secuencia muestra los objetos que intervienen en el escenario con líneas discontinuas verticales, y los mensajes pasados entre los objetos como vectores horizontales. Los mensajes se dibujan cronológicamente desde la parte superior del diagrama a la parte inferior; la distribución horizontal de los objetos es arbitraria. Durante el análisis inicial, el modelador típicamente coloca el nombre "business" de un mensaje en la línea del mensaje. Más tarde, durante el diseño, el nombre "business" es reemplazado con el nombre del método que está siendo llamado por un objeto en el otro. El método llamado, o invocado, pertenece a la definición de la clase instanciada por el objeto en la recepción final del mensaje.

Las modificaciones de los diagramas de secuencias tienden básicamente a permitir la reutilización de los diagramas, agregando los elementos de tipos Fragmento Combinado.

Es un diagrama que representa una interacción, poniendo el foco en la secuencia de los mensajes que se intercambian, junto con sus correspondientes ocurrencias de eventos en las líneas de vida.

- **Diagrama de comunicación (antiguos diagramas de colaboración)**

Anteriormente tenían el nombre de "diagramas de colaboración". Por ser las colaboraciones un diagrama de interacción, al igual que los diagramas de

secuencias, heredan la misma capacidad de soportar fragmentos combinados. En UML 2.0, un diagrama de comunicación es una versión simplificada del diagrama de colaboración de la versión de UML 1.x.

Un diagrama de comunicación modela las interacciones entre objetos o partes en términos de mensajes en secuencia. Los diagramas de comunicación representan una combinación de información tomada desde el diagrama de clases, secuencia y diagrama de casos de uso describiendo tanto la estructura estática como el comportamiento dinámico de un sistema.

Los diagramas de comunicación y de secuencia describen información similar y pueden ser transformados unos en otros sin dificultad.

Para mantener el orden de los mensajes en un diagrama de comunicación, los mensajes son etiquetados con un número cronológico y colocados cerca del enlace por el cual se desplaza el mensaje. Leer un diagrama de comunicación conlleva comenzar en el mensaje 1.0, y seguir los mensajes desde un objeto hasta el siguiente, sucesivamente.

Es un diagrama que enfoca la interacción entre líneas de vida, donde es central la arquitectura de la estructura interna y cómo ella se corresponde con el pasaje de mensajes. La secuencia de los mensajes se da a través de un esquema de numerado de la secuencia.

- **Diagrama de tiempos**

Un diagrama de tiempos o cronograma es una gráfica de formas de onda digitales que muestra la relación temporal entre varias señales y cómo varía cada señal en relación a las demás.

Un cronograma puede contener cualquier número de señales relacionadas entre sí. Examinando un diagrama de tiempos, se pueden determinar los estados, nivel alto o nivel bajo, de cada una de las señales en cualquier instante de tiempo especificado, y el instante exacto en que cualquiera de las señales cambia de estado con respecto a las restantes.

El propósito primario del diagrama de tiempos es mostrar los cambios en el estado o la condición de una línea de vida (representando una Instancia de un clasificador o un rol de un clasificador) a lo largo del tiempo lineal. El uso más común es mostrar el cambio de estado de un objeto a lo largo del tiempo, en respuesta a los eventos o estímulos aceptados. Los eventos que se reciben se anotan, a medida que muestran cuándo se desea mostrar el evento que causa el cambio en la condición o en el estado.

- **Diagramas de revisión de interacciones**

Es un diagrama que muestra cómo interactúan varios diagramas de interacciones. Este tipo de diagramas es muy útil para mostrar de qué manera distintos escenarios se combinan.

Los diagramas de revisión de la interacción enfocan la revisión del flujo de control, donde los nodos son interacciones u ocurrencias de interacciones. Las líneas de vida y los mensajes, no aparecen en este nivel de revisión.

4.2.4. Diagramas UML 2.0

La superestructura del UML es la definición formal de los elementos que componen el UML 2.0. Este se encuentra organizado en paquetes, que definen los elementos internos del UML y de qué manera se relacionan. El diseño interno del UML 2.0 se encuentra orientado a objetos.