Resumen TDP

Motivación

El objetivo de TDP es entender y comprender el rol que tiene el paradigma orientado a objetos en la resolución de problemas de mediana a gran escala; sus usos, beneficios y tecnologías asociadas.

La ingeniería de software es la producción de software de calidad. Muchos aspectos son los que definen la calidad del software. Una definición para conceptos generales es: *"La calidad es la suma de todos aquellos aspectos o características de un producto o servicio que influyen en su capacidad para satisfacer las necesidades, expresadas o implícitas"* (ISO 8402). Las que se aplican al software son: *"Grado con el cual el cliente o usuario percibe que el software satisface sus expectativas."* (IEEE 729-83) y *"Capacidad del producto software para satisfacer los requerimientos establecidos"* (DoD 2168).

Que es lo que hace a un diseño, un buen diseño

- ¿Qué Clases vamos a tener?
- ¿Qué atributos?
- ¿Qué operaciones?
- ¿Qué relaciones?
- ¿Qué responsabilidades?
- Si tengo una clase, ¿Por qué esta esa clase ahí? ¿Cuál es el rol de esa clase? ¿Qué pasa si saco esa clase? ¿Con cuantas clases se comunica esa clase?

Si el software (de manera interna) está ordenado, es fácil agregar cambios, por lo tanto, posee un buen diseño. El **diseño** es el proceso a través del cual nosotros creamos la solución a un problema. Un buen diseño facilita la futura modificación del código.

El diseño nos obliga a hacernos preguntas sobre lo que estamos haciendo. Esas preguntas plantean nuevas dudas, correcciones y ahorra dinero en correcciones (son menos costosos reparar errores en la etapa de diseño que reparar errores en la etapa de desarrollo).

Todos queremos que nuestro sistema de software sea rápido, confiable, fácil de usar, legible, modular, estructurado, etc. La calidad del software puede dividirse en dos categorías:

En "usuarios" deberíamos incluir no solo a las personas que realmente interactúan con los productos finales.

- Interna: perceptibles solo para los profesionales de la informática que tienen acceso al código.
- Externa: Estamos considerando cualidades como la velocidad o la facilidad de uso, cuya presencia o ausencia en un producto de software puede ser detectada por sus usuarios.

Aquí están los factores de calidad externos más importantes, cuya búsqueda es la tarea central de la construcción de software orientado a objetos:

- Correctitud: *es la capacidad de los productos de software para realizar sus tareas exactas, según lo definido por su especificación*. La correctitud es la cualidad principal. Si un sistema no hace lo que se supone que debe hacer, poco importa todo lo demás. Incluso el primer paso hacia la correctitud ya es difícil: debemos ser capaces de especificar los requisitos del sistema de forma precisa. Un sistema de software serio toca tantas áreas que sería imposible garantizar su correctitud tratando todos los componentes y propiedades en un solo nivel. Solo nos preocupamos de *garantizar* que cada capa sea correcta, en el supuesto de que el nivel inferior sea correcto.
- Robustez: *es la capacidad de los sistemas de software para reaccionar adecuadamente ante acontecimientos anormales*. La robustez complementa la correctitud. La correctitud aborda el comportamiento de un sistema en los casos cubiertos por su especificación; la robustez caracteriza lo que sucede fuera de esa especificación. La robustez es, por naturaleza, una noción más confusa que la correctitud. Dado que aquí nos ocupamos de casos no cubiertos por la especificación. El papel de los requisitos de robustez es asegurarse de que, si se presentan tales casos, el sistema no provoque eventos catastróficos.

- Extensibilidad: *es la facilidad de adaptar los productos de software a los cambios de especificación*. Un gran sistema de software a menudo se ve a menudo como un gigantesco castillo de naipes en el que sacar cualquier elemento puede hacer que todo el edificio se derrumbe. Dos principios son esenciales para mejorar la extensibilidad:

- *Simplicidad de Diseño*: una arquitectura simple siempre será más fácil de adaptar a los cambios que una compleja.
- *Descentralización*: cuanto más autónomos sean los módulos, mayor será la probabilidad de que un simple cambio afecte solo a un módulo, o a una pequeña cantidad de módulos, en lugar de desencadenar una reacción en cadena de cambios en todo el sistema.

- Reusabilidad: *es la capacidad de los elementos de software para servir en la construcción de muchas aplicaciones diferentes*. La necesidad de reusabilidad proviene de la observación de que los sistemas de software a menudo siguen patrones similares; debería ser posible explotar esta similitud y evitar reinventar soluciones a problemas que se han encontrado antes. La reusabilidad influye en todos los demás aspectos de la calidad del software, ya que resolver el problema de la reusabilidad significa esencialmente que se debe escribir menos software y, por lo tanto, se puede dedicar más esfuerzo a mejorar los otros factores.
- Compatibilidad: *es la facilidad de combinar elementos de software con otros*. La compatibilidad es importante porque no desarrollamos elementos de software en un vacío: necesitan interactuar entre sí. Pero con demasiada frecuencia tienen problemas para. La clave de la compatibilidad radica en la homogeneidad del diseño y en acordar convenciones estandarizadas para la comunicación entre programas. Los enfoques incluyen:
  - *Formatos de archivo estandarizados.*
  - *Estructuras de datos estandarizadas.*
  - *Interfaces de usuario estandarizadas*.
- Eficiencia: *es la capacidad de un sistema de software para exigir la menor cantidad posible de recursos de hardware*. La comunidad de software muestra dos actitudes típicas hacia la eficiencia:
  - Algunos desarrolladores tienen una obsesión con los problemas de rendimiento, lo que los lleva a dedicar muchos esfuerzos a supuestas optimizaciones.
  - Pero también existe una tendencia general a restar importancia a las preocupaciones por la eficiencia.

  La eficiencia no importa mucho si el software no es correcto

  Los desarrolladores a menudo han mostrado una preocupación exagerada por la microoptimización. La preocupación por la eficiencia debe equilibrarse con otros objetivos como la extensibilidad y la reutilización; las optimizaciones extremas pueden hacer que el software sea tan especializado que no sea apto para cambios y reutilizaciones. La mejora constante en el poder de la computadora, por impresionante que sea, no es una excusa para pasar por alto la eficiencia, al menos por tres razones:

  - Alguien que compra una computadora más grande y más rápida quiere ver algún beneficio real de la potencia adicional.
  - Uno de los efectos más visibles de los avances en el poder de las computadoras es en realidad aumentar la ventaja de los buenos algoritmos sobre los malos.
  - En algunos casos, la eficiencia puede afectar la corrección.
- Portabilidad: *es la facilidad de transferir productos de software a varios entornos de hardware y software*. La portabilidad aborda variaciones no solo del hardware físico sino más en general de la **máquina hardware-software**.
- Facilidad de uso: *es la facilidad con la que las personas de diversos antecedentes pueden aprender a usar productos de software y aplicarlos para resolver problemas*. La definición insiste en los distintos niveles de experiencia de los usuarios potenciales. Este requisito plantea uno de los principales desafíos para los diseñadores de software: cómo brindar orientación y explicaciones detalladas a los usuarios novatos. Una de las claves para la facilidad de uso es la simplicidad estructural. Un sistema bien diseñado, construido de acuerdo con una estructura clara y bien pensada, tenderá a ser más fácil de aprender y usar que uno desordenado

  Principio de diseño de la interfaz de usuario

  No finja que conoce al usuario.

- Funcionalidad: *es el alcance de las posibilidades que ofrece un sistema*. Uno de los problemas más difíciles que enfrenta un líder de proyecto es saber cuánta funcionalidad es suficiente. Un buen producto software se basa en un pequeño número de ideas poderosas; incluso si tengo muchas características especializadas, todas deberían ser explicables como consecuencia de estos conceptos básicos.
- Puntualidad: *es la capacidad del sistema de software para ser lanzado cuando sus usuarios lo deseen o antes*. La puntualidad es una de las frustraciones de nuestra industria. Un gran producto de software que aparece demasiado tarde puede perder su objetivo por completo. La puntualidad sigue siendo, para grandes proyectos, un fenómeno poco común.
- Verificabilidad: *es la facilidad de preparar los procedimientos de aceptación*.
- Integridad: *es la capacidad de los sistemas de software para proteger sus diversos componentes contra el acceso y la modificación no autorizados*.
- Reparabilidad: *es la capacidad de facilitar la reparación de los defectos*.
- Economía: compañera de la puntualidad, *es la capacidad de un sistema para completarse dentro o por debajo del presupuesto asignado*.

En una lista de factores de calidad del software, uno podría esperar encontrar la presencia de una buena documentación como uno de los requisitos. Pero esto no es un factor de calidad separado; en cambio, la necesidad de documentación es una consecuencia de los otros factores de calidad vistos anteriormente.

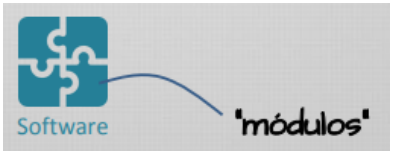
Externa	Interna
Permite a los usuarios comprender el poder de un sistema y usarlo convenientemente, es una consecuencia de la definición de facilidad de uso.	Permite a los desarrolladores de software comprender la estructura y la implementación de un sistema, es una consecuencia del requisito de extensibilidad.

En lugar de tratar la documentación como un producto separado del software propiamente dicho, es preferible hacer que el software se autodocumente tanto como sea posible. Esto se aplica todo tipo de documentación:

- Al incluir funciones de "ayuda" en línea y adherirse a convenciones de interfaz de usuario claras y consistentes, alivia la tarea de los autores de manuales de usuario y otras formas de documentación externa.
- Un buen lenguaje de implementación eliminará gran parte de la necesidad de documentación interna si favorece la claridad y la estructura.

Diseño Modular

¿Qué significa un **buen diseño** de software? Hay muchas características que un buen diseño debería cumplir. Principalmente, observar una buena organización en **módulos**.



Los módulos son elementos que se *focalizan* en un aspecto del sistema, *intercambiables* e *independientes*. A lo que nos referimos cuando hablamos de un buen diseño modular, es que estos módulos (en el POO, las clases) están bien definidos, tengas responsabilidades y funciones claras.

Las características que deben tener un buen diseño modular, según Meyer, son:

- Pocas interfaces: todo modulo debe comunicarse con pocos módulos.
- Interfaces pequeñas: si dos módulos se comunican, deben intercambiar poca información.
- Interfaces explicitas: la forma de comunicación entre dos módulos debe ser obvia (nombres de las clases, nombre de las operaciones).
- Ocultamiento de información: ayuda a minimizar las interfaces. Cada módulo posee métodos públicos y métodos ocultos. Ayuda a que las otras clases desconozcan la implementación de otra clase.

Documentación de la interfaz del módulo

Permite a los desarrolladores de software comprender las funciones proporcionadas por un módulo sin tener que comprender su implementación.

- Mapeo directo: la estructura modular del sistema tiene que ser compatible con aquellos elementos generados en el proceso de modelamiento.

La programación orientada a objetos **favorece un buen diseño modular**. Propone una forma de observar el mundo real, sus elementos y sus relaciones

Debe satisfacer:

- Descomposición modular: Ayuda a descomponer el problema en subproblemas.
- Composición modular: Favorece la producción de elementos de software que pueden combinarse.
- Comprensión modular: Ayuda a producir software en el cual el humano puede comprender cada módulo.
- Continuidad modular: Favorece que un cambio pequeño en la especificación impacte en uno o pocos módulos.
- Protección modular: Favorece la contención de situaciones anormales en ejecución.

Historia

En 1970 empezaba a aparecer la idea de la computadora personal. Desde el punto de vista de la programación, en este momento, no existía la programación ni el rol del programador. En los 70, cada profesional, que tenía acceso a la computadora, encontraba una herramienta que le podría ayudar en su trabajo. Ellos aprendían a programar y a realizar sus propios programas.

A medida que pasa el tiempo, se empiezan a querer resolver problemas más complejos y al intentarlo se dieron cuenta que sus conocimientos no eran suficientes, que necesitaban a alguien más dedicado a programar. Con el tiempo, fue necesario un equipo de personas dedicadas a programar.

Los primeros procesos ingenieriles que se aplicaron a la construcción del software, imitaban los de la construcción física. Estos procesos no dieron resultado. Al inicio se dieron cuenta que el software era muy difícil de entender, cuáles eran los requerimientos y que las condiciones en las cuales se desarrollaba o se empezaba a trabajar el software cambiaban mucho. Entonces, se dieron cuenta que era necesario poder adaptarse a esos cambios y que era necesario contar con lenguajes de programación que acompañaran este proceso.

Finalmente, los procesos de construcción de software pretendieron imitar un árbol. Lo que se busco fue poder trabajar con el software con un sistema de adaptación como se trabajaría con un árbol. Pensando en la necesidad de reutilizar cosas y flexibilidad a cambios. La gente se fue dando cuenta que necesitaban una forma distinta de pensar no solo la implementación sino también la solución a los problemas. Ante esta necesidad es que aparece el paradigma orientado a objetos (Una forma de pensar).

El paradigma orientado a objetos nos brinda:

- Extensibilidad.
- Confiabilidad.
- Reusabilidad.

El paradigma no es la solución perfecta a todos los problemas. No todos los problemas se solucionan con POO

Paradigma Orientado a Objetos

Cuando hablamos del paradigma orientado a objetos, lo primero en venir a la mente es el concepto de clase. Lo que introduce el POO es el enfoque en los sustantivos. Después debemos enfocarnos en las características de esos sustantivos, **atributos**, y en las acciones que realiza ese objeto, **mensajes**.

Los tres *conceptos claves* en la orientación a objetos forman la estructura estática y dinámica de todo software OO:

- Los **objetos** son representaciones de los objetos del mundo real o conceptos. Empaquetan datos y operaciones.
- Los **mensajes**, por medio de los cuales los objetos se comunican entre sí.
- Las **clases**, que definen la estructura y el comportamiento de los objetos.

Java une esos dos conceptos.

Otros leguajes de programación, los separan.

Una clase, en POO, es un tipo de dato abstracto y es un módulo. Es un tipo de dato abstracto ya que define un tipo de dato y un conjunto de operaciones sobre ese tipo de dato. Es un módulo ya que la clase funciona como una unidad de contención de información, que no es lo mismo que un tipo de dato abstracto.

Un concepto, dependiendo el problema, puede tener una representación o no. Si a ese concepto, en un problema dado, se le da mucha importancia, muy probablemente se considere una clase.

Toda clase tiene tres elementos básicos:

- El nombre debe *representar a la abstracción del conjunto* de las instancias de la clase, y debe ser único en todo el sistema. Generalmente, está acompañada de un texto denominado responsabilidades.
- Los *atributos* describen la representación interna de los valores que son instancia de la clase.
- Los *métodos* o *servicios* son las operaciones que se pueden efectuar sobre las instancias de la clase.

Un atributo calculado es un atributo el cual su valor no se almacena, se calcula en ejecución

Una clase es un patrón o esquema que especifica los atributos y servicios compartidos por todos los objetos que pertenecen a ella. Al definir una clase, se está definiendo un concepto general. Un objeto es una instancia de una clase. Cada clase puede tener *una*, *varias* o *ninguna instancia*. Algunos objetos serán una *representación directa* de objetos del mundo real. Otros serán *representaciones de abstracciones* correspondientes a la solución del problema.

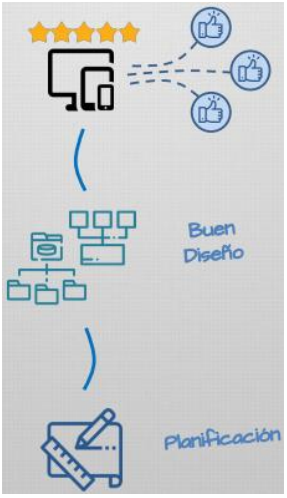
Los tres *mecanismos claves* en la orientación a objetos:

- **Encapsulado**, el mecanismo para empaquetar datos y procedimientos en los objetos. Me interesa que hace la clase y no como lo hace.
- **Polimorfismo**, la habilidad de implementar el mismo mensaje en formas diferentes en objetos diferentes.
- **Herencia**, el mecanismo para diseminar la información definida en clases genéricas a otras clases consideradas casos especiales de las anteriores.

El modelo de computación OO se basa en las siguientes reglas:

- En ejecución, un sistema de software está conformado únicamente por objetos.
- Todo objeto es **instancia de una clase**.
- En todo momento existe un único objeto en ejecución, denominado **instancia actual**.
- Toda computación se realiza cuando un objeto resuelve un **pedido** de un servicio de otro objeto.
- El objeto que llama se denomina **cliente**, y el que realiza la acción se denomina **servidor**.

Diseño Orientado a Objetos



En el proceso de desarrollo de software posee varias tareas a realizar:

- Descubrir qué quiere el cliente.
- Descubrir cómo lo quiere.
- Idear cómo debería ser el producto final.

El camino a la **excelencia** y la **calidad** siempre es un camino *planificado*, nunca improvisado.

Cuando hablamos de planificación, hablamos de un proceso por el cual yo puedo evaluar las características de algo grande y costoso a partir de algo chiquito y barato.

El proceso de la planificación, nos obliga a nosotros a hacernos preguntas sobre eso que queremos construir. También es un proceso en el cual nosotros descubrimos cosas sobre aquello que queremos construir.

¿Para qué modelar? Para proveer una **estructura para la solución** del problema, para proveer las **abstracciones necesarias** para lidiar con la complejidad, para **reducir los costos** de desarrollo y para **minimizar el riesgo** de cometer errores.

El proceso de modelado es el proceso de planificación de software. A través del modelado vamos a tener distintas vistas del mismo software, pero enfocándose en distintas características. El modelado es muy importante en el POO. Es importante conocer cuatro principios de modelado:

- La elección de qué modelo crear influye en cómo el problema es atacado y como la solución toma forma.
- Todo modelo puede ser expresado en diferentes niveles de precisión.
- Los mejores modelos están conectados con la realidad.
- Un solo modelo no es suficiente.

Si bien existen diversos **tipos de modelos**, los modelos gráficos son especialmente importantes en ingeniería. El Modelado orientado a objetos requiere un buen uso de **modelos gráficos**.

Igual que en la historia de la programación, el modelado fue evolucionando a medida que pasaba el tiempo. En un punto de su evolución, cada empresa que desarrollaba software poseía su propio diseño para modelar. Esto generaba un problema al momento de la comunicación o de cambio de empresa y llevó a la creación del diseño UML (Lenguaje Unificado de Modelado).

El UML es un **lenguaje gráfico para el diseño completo de sistemas**, el cual posee reglas. UML no es un único modelo, son muchos modelos. Estos modelos representan las distintas perspectivas del software.

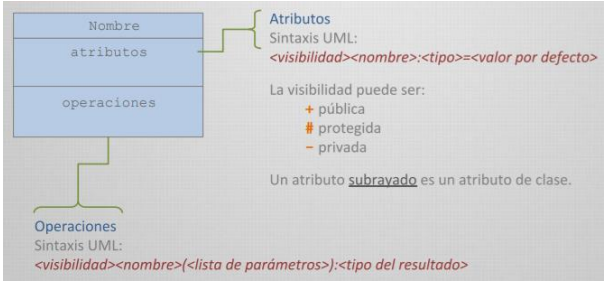
La especificación UML define varios tipos de diagramas. Ocho son los tradicionales desde las primeras versiones:

- *Diagramas de casos de uso.*
- *Diagramas de clase y objetos.*
- *Diagramas de Estados.*
- *Diagramas de secuencia.*
- *Diagramas de colaboración.*
- *Diagramas de actividades.*
- *Diagramas de componentes.*
- *Diagramas de despliegue (deployement).*

La perspectiva más conocida en UML es el diagrama de clases, donde se enfoca en la parte estática del software. El sistema de clases describe los tipos de objetos en el sistema y las dependencias estáticas entre ellos.

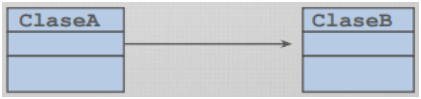
Más sobre diseño y UML

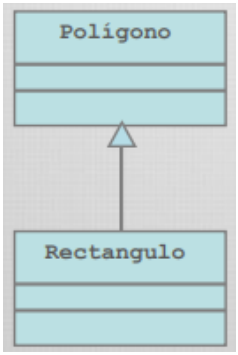
El diseño va acompañado de un diagrama o un conjunto de diagramas (UML). Cuando hablamos de un diagrama de clases, la representación principal que tenemos es la representación de una clase. Una clase se representa con un rectángulo con el nombre de la clase dentro. El gran beneficio que tiene esta etapa de diseño es que nosotros podemos trabajar con diferentes niveles de abstracción (distintos niveles de detalle).



En un momento inicial las clases en un diagrama de clases son simplemente rectángulos con los nombres de las clases dentro. A medida que se avanza en el análisis del problema se le van agregando detalles a las clases, como los tributos, de que tipo son los atributos y sus métodos.

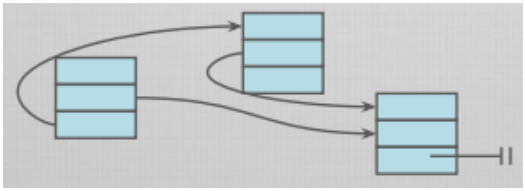
Las clases que conforman un sistema pueden relacionarse de diferentes maneras. Típicamente, si uno de los atributos de una clase A es de clase B, *las clases A y B* están **asociadas**. Una de las clases "conoce" a la otra y puede comunicarse con ella. La Clase A tiene un atributo de tipo Clase B. Esas relaciones tienen un nombre, tiene una cardinalidad y una dirección (¿Quién se asocia con quién?). La asociación denota una *conexión estructural* entre objetos.





Algunas relaciones entre objetos sugieren más que un conocimiento circunstancial entre ellos. Se denomina **agregación** a un tipo especial de asociación que implica una fuerte dependencia estructural ("es parte de"). La **composición** es una relación donde el objeto compuesto controla la creación de los objetos que lo componen. Los objetos que son parte del todo, no pueden existir separadamente. Una clase puede ser una **extensión** de otra clase, puede ser una **especialización** de otra clase o puede ser una **combinación** de otras clases. La herencia es un **componente central** en la orientación a objetos.

Una **referencia** es un valor que puede estar nulo o asociado. Si la referencia está asociada, entonces identifica un único objeto entre los demás que componen el sistema. Usaremos flechas para indicar referencias asociadas a otros objetos en el sistema.



En la **semántica por referencia**, la creación de objetos es *explícita*, por medio de algún operador que requiere la invocación de un constructor. Efectos de la creación explícita de un objeto:

1. Crear una instancia de la clase correspondiente.
2. Inicializar cada campo de la nueva instancia acordé a los valores por defecto.
3. Ejecutar la implementación del constructor.
4. Asociar una referencia a este objeto recién creado.

El manejo por referencia provee:

- Posibilidad de **compartir** información.
- Modelado de la relación "*conoce*" o "*tiene acceso a*" entre objetos.
- Manejo de **agrupación de datos** con identidad.
- **Eficiencia** en el espacio de memoria.

Problemas:

- *Dynamic aliasing.*
- Posibilidad de referencias colgadas.
- Problema de la persistencia de objetos.

Operaciones clásicas por referencia:

- Comparación de referencias: la comparación de referencias es una operación que devuelve verdadero, si la referencia es idénticamente la misma, falso en caso contrario.
- Clonación: es una operación de *creación e inicialización* en base a un objeto modelo. Se realiza por medio de una función predefinida: "Nuevo = viejo. Clone()". No todos los lenguajes poseen una función predefinida de clonación.
- Copia: la copia es una operación entre dos referencias, en la que los campos de una toman los valores provistos en los campos de otra.
- Comparación de objetos: la comparación de objetos es una operación entre dos referencias, que devuelve verdadero si los campos de ambos objetos referenciados contiene exactamente los mismos valores, aunque se trate de diferentes objetos. Falso en caso contrario. Es una comparación *superficial*.
- Clonación y comparación en profundidad: son la asociación operaciones correspondientes aplicadas recursivamente a todos los niveles de referencias entre objetos.

En la **semántica por valor**, la creación de objetos puede ser *implícita*. El constructor es invocado implícitamente al momento de la creación del objeto por valor. Un objeto puede literalmente contener a otro objeto. Este último se denomina **subobjeto** del primero.

- La noción de subobjeto implica que el contenido solo puede ser conocido por el objeto continente.
- Es posible sin embargo que el subobjeto referencia a otros objetos que no son el que lo contiene.
- Una entidad con semántica por valor no puede ser nula.

- El tiempo de vida del subobjeto es el tiempo de vida del objeto que lo contiene.
- La construcción de un objeto involucra implícitamente la construcción de todos los subobjetos que contiene.

Efectos de la creación implícita de objetos:

1. Inicializar cada campo de la nueva instancia acorde a los valores por defecto.
2. Ejecutar la implementación del constructor.

El manejo por valor provee:

- Manejo de **agregación** de datos.
- Modelado de la relación "*conoce*" o "*es parte de*" entre objetos.
- Eficiencia en el tiempo de acceso.

Problemas:

- Duplicación de información.

Es posible asociar o comparar entidades con semánticas diferentes. Se denominan **operaciones híbridas.**

UML provee también diagramas para reflejar la estructura dinámica firmada por objetos en algún momento de la ejecución. El diagrama de objetos muestra las instancias de las clases.

Los **diagramas de interacción** son modelos que describen como un grupo de objetos *colabora para realizar una tarea particular*. Hay dos tipos de diagramas de interacción:

- **Diagrama de secuencia.**
- **Diagrama de colaboración**.

Diseño, diseño

Un buen diseño tiene como característica el mapeo directo. Aquellos elementos que forman parte de mi sistema tienen un mapeo con los elementos que forman parte del problema.

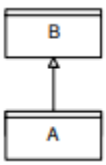
Herencia

Teniendo en cuenta la reutilización y la extensibilidad es que se crea la herencia. No se puede considerar tener un paradigma orientado a objetos sin tener herencia. Un concepto que va de la mano de la herencia, el reúso y de la extensibilidad es la **generalización**. La generalización busca detectar elementos comunes a un conjunto de elementos y representar esas características comunes en un concepto más general que permite describir los demás conceptos.

La genericidad es un tipo particular de generalización

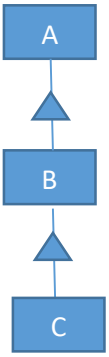
La herencia es un tipo de relación que establece que todo lo que yo tengo en la clase más alta de la jerarquía de herencia va para abajo, todo lo que yo diga en función de la jerarquía más alta va a ser válido para todas las cosas debajo en la jerarquía de herencia. Además, la clase que hereda puede *agregar elementos propios* que se suman a los herederos. Es posible en la clase que hereda *cambiar la implementación* de una operación por otra más conveniente. Esto se denomina **redefinición de operaciones**. A veces es necesario cambiar la implementación de ciertas operaciones heredadas. Para ello se vuelve a declarar la operación en la clase hija, y se provee una nueva implementación. El *signature* del servicio debe ser el mismo.

Más sobre herencia



Sea una relación como la de la izquierda, la Clase B es la clase “Padre”, “Superclase” o “Base”. La Clase A es la clase “Hija”, “Subclase” o “Derivada”. En este caso se dice que la clase Hija es un descendiente directo de la clase Padre y la clase Padre es ancestro directo de la clase Hija. Esto da lugar a lo que llamamos **jerarquía de herencia**.

En una jerarquía de herencia, existe la herencia múltiple. Cuando una clase posee más de una clase padre.



Según el diagrama de la derecha, todo lo que está encima de la clase B (con B incluido) son ancestros de B. A su vez, todo lo que está debajo de B (con B incluido) son los descendientes de B. Los ancestros/descendientes propios de B, son aquellas clases que se encuentran encima/debajo de B, sin incluir a B.

Las instancias de una clase son los objetos que son instancias de algún descendiente de la clase. Cuando hablemos de las instancias de la clase padre, también vamos a estar hablando de las instancias de las clases hijas.

Siempre nos mantenemos dentro de lo que establece la jerarquía de herencia

La jerarquía de herencia también da lugar al polimorfismo ("poli" = muchas, "morfismo" = formas). Una **asociación polimórfica** ocurre cuando a una referencia de una clase se le asocia una referencia a una instancia no directa. ¿Puede cambiar su forma libremente? No, lo tiene que hacer bajo ciertas reglas, de eso se encarga el chequeo de tipos.

El control de la validez de las asignaciones se realiza como parte del **chequeo de tipos**. Este proceso verifica el programa de acuerdo a un conjunto de reglas definidas en el sistema de tipos. Los objetos requieren declaración previa del tipo de dato al que pertenecen.

¿Cuáles son las ventajas de tener que declarar los tipos de objetos?

- Nuestro software será más confiable.
- Nuestro software es más fácil de leer.
- Nuestro software es más eficiente.

"Un lenguaje orientado a objetos es **estáticamente tipado** si está equipado con un conjunto de *reglas de consistencia*, controlada por los compiladores, cuya observancia por el texto del software garantiza que la ejecución del software no incurrirá en una violación de tipos" Meyer.

Conformidad de tipos: un tipo U *conforma* un tipo T solo si el tipo base de U es un descendiente de la clase base de T; para tipos derivados genéticamente, cada parámetro actual de U debe conformar el parámetro formal de T.

Dado que una referencia puede estar asociada a objetos de diferentes tipos, podemos hacer dos distinciones del "*tipo de una referencia*":

- El **tipo estático** de una entidad x es el tipo usado para declarar esa entidad.
- El **tipo dinámico** de una entidad x en un determinado momento de ejecución es la clase de la que es instancia directa el objeto asociado a x en ese momento.

¿Qué es un tipo de dato? Un **conjunto de valores** y las **operaciones** que pueden aplicarse sobre esos valores. El tipo A es un subtipo del tipo B cuando la especificación de A implica la especificación de B. Es decir, cualquier objeto que satisfaga la especificación de A también satisface la especificación de B, porque la especificación de B es más débil.

Subclase: indica que la nueva clase se hereda de la clase principal

Subtipo: Se enfatiza que la nueva clase tiene el mismo comportamiento que la clase principal

La relación entre tipos y subtipos y la separación con la implementación lleva a dos *concepciones diferentes de herencia*:

- **Interfaz**
- **Implementación**

Es una diferencia sutil, pero importante. Para eso es necesario comprender la diferencia entre la *clase* de un objeto y su *tipo*:

- La **clase** define *como el objeto es implementado* (Estado interno e implementaciones concretas)
- El **tipo** solo se refiere a la *interfaz*.

Por esta razón existe una diferencia formal entre *herencia de interfaz* y *herencia de clase o implementación*:

- La **herencia de interfaz** describe cuando un objeto puede ser usado en lugar de otro, porque al fin y al cabo, *responden a los mismos pedidos*.
- La **herencia de clase** define la implementación de un objeto en términos de la implementación de otro objeto.

A veces se confunden porque muchos lenguajes no hacen una distinción explícita.

Métodos y herencia

Una operación que decimos que está, pero no la implementamos, se la llama operación abstracta. Si una clase tiene por lo menos una operación abstracta, decimos que esa clase es abstracta. ¿Qué quiere decir que una clase sea abstracta? Significa que nosotros no podemos crear instancias de esa clase.

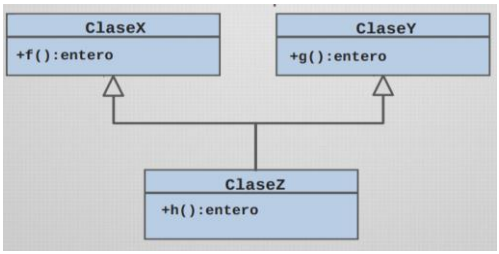
Es posible tener una clase abstracta con todos sus métodos implementados. ¿Por qué razón haría eso? Yo no quiero que existan instancias de esa clase, en mi problema no debería tener instancias de esa clase.

Si se le agrega un asterisco a la derecha de una clase o de un método (en el diagrama hecho a mano) para indicar que es abstracto. En el caso de que el diagrama esté hecho de manera digital, se escribe el nombre de la clase o el nombre del método con *itálica*.

En contrapartida, las clases completamente definidas se denominan **clases concretas**.

Una clase que hereda de una clase abstracta puede incluir sus propias implementaciones de las operaciones abstractas. En ese caso se dice que la operación es **efectiva** en la clase hija. Si una clase hija define un método abstracto de la clase padre, entonces se señala ese método con un más a la derecha.

Si se efectivizan todas las operaciones, entonces la clase hija es una clase concreta. Si se deja alguna operación sin efectivizar, permanece abstracta y la clase hija es también una clase abstracta.



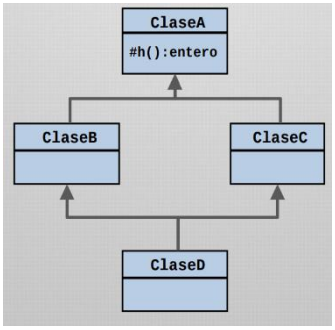
Si una clase hija redefine un método de la clase padre, se escribe el nombre del método.

En la herencia múltiple una clase puede heredar de dos o más clases. La idea de heredar es exactamente la misma que en el caso de la herencia simple. Sin embargo, algunas situaciones pueden ser problemáticas.

Cuando dos clases padre poseen dos métodos con el mismo nombre y parámetros. En estas situaciones se dice que existe una **ambigüedad de nombres** o **colisión de nombres**. La colisión se produce cuando los dos servicios son efectivos. Cuando uno de ellos es diferido (abstracto) no se produce una colisión pues se considera al otro como una efectivizar.

Los lenguajes pueden optar por dos caminos:

- Proveen algún mecanismo para *tratar la ambigüedad* (Permitir renombrar un atributo u operación en el momento en que es heredado, mantener dos versiones y explicitar en la llamada cuál de las operaciones se desea invocar).
- Proveen algún *comportamiento por defecto* (establecer prioridades)



Otro de los problemas que acarrea la herencia múltiple es la denominada **herencia repetida**, también conocido como el *problema del diamante*. La herencia repetida surge cuando una clase es descendiente de otra en más de una forma.

En este caso, el resultado de la herencia es *una sola copia del atributo* u operación en la clase descendiente. Esto se denomina **compartir** (**sharing**). Sin embargo, a veces la entidad heredada tiene un significado diferente por cada posible camino.

Necesitamos la habilidad de mantener *dos copias separadas* del mismo atributo heredado de formas diferentes. Cuando un atributo se hereda en forma repetida bajo diferentes nombres, el resultado de la herencia es la **replicación** del atributo u operación en la clase descendiente.

Si los atributos o servicios son verdaderos con el mismo nombre:

1. Si una de las versiones es concreta, y las demás abstractas, entonces la versión concreta efectiviza a las demás.
2. Si ambas versiones son concretas, pero el servicio es recibido todas las versiones son unificadas en la nueva versión.
3. Si hay más de una versión concreta, pero no hay redefinición, entonces ocurre una colisión de nombres.

No siempre es fácil identificar cuándo utilizar la herencia. Por lo general conviene observar la relación entre clases. La más frecuente es la relación "*es un*". Sin embargo, no hay que confundir con la relación instancia-clase. Asociamos la herencia con la relación "es un" cuando hablamos de *clases como agrupaciones de objetos*.

Bertrand Meyer distingue algunos tipos generales de herencia, que califica como "usos válidos" de esta técnica. Tipos de herencia:

- **Herencia de modelo**: representa la relación "es un" entre las abstracciones del modelo (herencia de extensión, de restricción, de subtipo, etc.).
- **Herencia de software**: representa relaciones en el software, sin vinculación necesaria con el modelo. (Herencia de implementación, de facilidades, etc.)
- **Herencia de variación**: representa la descripción de una clase de acuerdo a las diferencias con otra clase. (Variación funcional, variación de tipos)

El polimorfismo da lugar a la **vinculación dinámica de código**. Si al declarar una variable, el tipo estático es de una clase ancestros propios de otra clase y el tipo dinámico es de un descendiente propio de esa misma clase, al llamar a un método se busca que método ejecutar desde la clase del tipo dinámico hacia arriba. Esto es la vinculación dinámica de código.

La redefinición no debe *traicionar* a los clientes cuando hay acceso polimórficos.

En compilación, el compilador se fija si el método (definido o no) que quiere llamar pertenece a la clase del tipo estático. En ejecución se va a ejecutar el método definido más próximo a la clase de tipo dinámico.

Más sobre métodos y herencia

El paradigma nos brinda una herramienta para que una clase que redefine un método ya definido en un ancestro, pueda utilizar el método del ancestro e ignore el método redefinido que posee la clase. Esa herramienta es "**super.metodo**", esta herramienta posee la limitación que solo nos deja revisar el ancestro anterior. Si la clase no redefine el método a utilizar, pero se utiliza la herramienta super, se va a buscar el método en el ancestro.

La herramienta super no se puede concatenar con más super.

Concurrencia

La concurrencia es una nueva forma de pensar cómo programar. Para poder empezar a entender cómo funciona la concurrencia primero tenemos que ver cómo funciona nuestro código cuando lo ejecutamos.

Una de las posibilidades para implementar la concurrencia es crear clases que implementen la interfaz Runnable.

Cuando nosotros ejecutamos un programa, aparece lo que es el hilo de ejecución, por defecto siempre tenemos uno (programa secuencial). Es posible tener más de un hilo de ejecución, esta posibilidad se llama **concurrencia**.

Otra posibilidad para trabajar con múltiples hilos es crear clases que extienden de Thread

Dos procesos en ejecución, correspondientes a dos programas diferentes. Existe un *instruction pointer para cada uno de ellos*. Una vez que uno de los hilos termina, no vuelve al punto en el hilo anterior de donde nace, sino que este "muere". Dos procesos en ejecución, correspondientes a un *mismo programa*. Existe un *instruction pointer para cada uno de ellos*.

La concurrencia tiene muchas particularidades y mucho poder. No hay que abusar de los hilos ya que es costoso para el sistema operativo.

Hablar de concurrencia no es lo mismo que hablar de paralelismo. Concurrencia es ejecutar simultáneamente varios programas. Paralelismo es la ejecución de varios procesadores.

Nos puede pasar que al emplear concurrencia, 2 o más hilos estén trabajando con una misma variable global, esto nos lleva a posibles problemas en el código. Hay formas de solucionar estos problemas, lo importante es identificar si la situación en la que estamos puede ser crítica y definir zonas seguras (Thread-safety), implica que el código es *seguro de ser utilizado por diferentes hilos* de ejecución, manteniendo la consistencia pretendida.

La **sincronización es necesaria** para los datos compartidos y asegurar la consistencia. Existen requerimientos especiales, como:

- Los datos compartidos deben accederse de forma atómica.
- Un proceso puede "reservar" un dato para su uso.
- Las operaciones deben esperar si los datos están en un estado incorrecto.

La concurrencia requiere especial atención en algunos aspectos:

- **Seguridad**: los procesos concurrentes pueden manipular equivocadamente datos compartidos.
- **Ciclo de vida**: un proceso puede "esperar eternamente" si no es manipulado correctamente.
- **No determinismo**: el mismo programa puede no devolver el mismo resultado al ejecutarse concurrentemente.
- **Tiempo de ejecución**: la coordinación, el cambio de contexto, y la sincronización, llevan tiempo.

Java permite programación *multithread*. Una operación de un objeto puede ejecutarse en threads diferentes, bajo memoria compartida. Existen dos formas de obtener objetos con código concurrente:

- **Heredar de la clase Thread**:
  - Contra: tendrá todos los métodos de la clase Thread.
  - Pro: probablemente sea la mejor abstracción: un objeto es un Thread.
- **Implementar la interfaz Runnable para un objeto de tipo Thread:**
  - Pro: a veces heredar de Thread es impracticable.
  - Contra: menos simple.

Principios de diseño

La calidad de las cosas que nosotros hacemos, no depende de la herramienta que nosotros usemos sino de nuestra habilidad para crear software. En este sentido es que nacen los principios de diseño, conocidos como los **principios S.O.L.I.D.** (son principios que guían el buen diseño de software).

¿Por qué principios y no reglas? Porque un principio es más una sugerencia, propuesta, dirección en la que se debe ir; es que hay que hacerlo sí o sí.

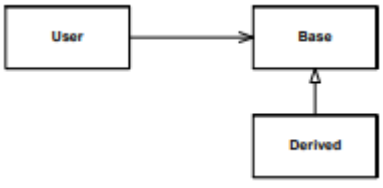
Los 5 principios son:

- **S**: Single-responsability principle. Lo que nos dice este principio es que cada clase debería tener una responsabilidad única, una razón por la cual está. Un cambio en las especificaciones, debería derivar en un cambio en solo UNA CLASE.

Intenta evitar el uso de la “Clase Dios”

- **O:** Open-closed principel. Una clase debería ser abierta para ser extendida pero cerrada para hacer modificaciones. Los módulos que cumplen este principio exhiben dos cualidades:
  - *Abierto para extensión*: el módulo puede ser extendido para comportarse de otra forma, según cambios en los requerimientos.
  - *Cerrados para modificación*: no se admiten cambios en el código, ni son necesarios.

  Existen varias técnicas para lograrlo a gran escala. Todas estas técnicas se basan en la abstracción.
  - **Polimorfismo dinámico.**
  - **Polimorfismo estático.**
  - **Metas Arquitectónicas del OCP**
- **L:** Liskov substitution principle. Las subclases deben ser sustituibles por sus clases base. Esto implica:
  - Para los métodos:
    - Debe haber un método en el subtipo por cada método en el supertipo.



        - Los métodos del subtipo no requieren nada adicional para funcionar.
        - Los métodos del subtipo no ofrecen menos que los del supertipo.
    - Para las propiedades:
        - Todas deben ser garantizadas por el supertipo.

  Esto puede parecer obvio, pero hay sutilezas que deben tenerse en cuenta:
    - **El dilema círculo/elipse.**
    - **Los clientes arruinan todo.**
    - **Diseño por contrato.**
    - **Repercusiones de la violación de LSP.**
- **I:** Interface segregation principle. Muchas interfaces específicas del cliente son mejores que una interfaz de propósito general. La esencia del principio es bastante simple. Si tiene una clase que tiene varios clientes, en lugar de cargar la clase con todos los métodos que necesitan los clientes, cree interfaces específicas para cada cliente.
- **D:** Dependency inversión principle. Depende de las abstracciones. No dependa de las concreciones. La inversión de dependencia es la estrategia de depender de interfaces o funciones y clases abstractas, en lugar de funciones y clases concretas.

Estos principios van de la mano de los objetivos del paradigma: favorecer la flexibilidad, la reusabilidad, la seguridad, la extensibilidad. Estos principios buscan enfatizar estos objetivos que tenía el paradigma. Estos principios nos ayudan a pensar, a decidir cómo organizar las clases, nuestras jerarquías, nuestras asociaciones.

Principios de la arquitectura de paquetes

¿Cómo elegimos qué clases pertenecen a qué paquetes? A continuación se presentan tres principios conocidos como Principios de Cohesión de Paquetes, que intentan ayudar al arquitecto de software:

- **El principio de equivalencia de liberación y reutilización (REP)**: *El gránulo de reutilización es el gránulo de liberación.* Un elemento reutilizable, no se puede reutilizar a menos que esté gestionado por un sistema de liberación de algún tipo. Por lo tanto, un criterio para agrupar clases en paquetes es la reutilización. Dado que los paquetes son la unidad de liberación, también son la unidad de reutilización.
- **El Principio de Cierre Común (PCC):** *Clases que cambian juntas, pertenecen juntas.* Agrupamos las clases que creemos que cambiarán juntas. Esto requiere una cierta cantidad de precisión ya que debemos anticipar los tipos de cambios que son probables.
- **El Principio Común de Reutilización (PRC):** *Las clases que no se reutilizan juntas no deben agruparse.*

Estos tres principios son mutuamente excluyentes. Eso es porque cada principio beneficia a un grupo diferente de personas. El REP y el CRP facilitan la vida de los reutilizadores, mientras que el CCP facilita la vida de los mantenedores. El PCC se esfuerza por hacer paquetes lo más grandes posible. El CRP, sin embargo, trata de hacer paquetes muy pequeños.

Los siguientes tres principios gobiernan las interrelaciones entre paquetes:

- **El Principio de Dependencias Acíclicas (ADP):** *Las dependencias entre paquetes no deben formar ciclos.*
    - **Un ciclo se arrastra**. Esto significa que alguien debe observar la estructura de dependencia del paquete con regularidad y romper los ciclos dondequiera que aparezcan. De lo contrario, las dependencias transitivas entre módulos harán que cada módulo dependa de todos los demás módulos.
    - **Rompiendo un Ciclo**. Los ciclos se pueden romper de dos maneras. El primero implica la creación de un nuevo paquete y el segundo hace uso del DIP y el ISP.
- **El Principio de Dependencias Estables (SDP):** *Depende de la dirección de la estabilidad.*
    - **Estabilidad.** La estabilidad está relacionada con la cantidad de trabajo requerido para hacer un cambio. Hay muchos factores que hacen que un paquete de software sea difícil de cambiar. Una forma segura de hacer que un paquete de software sea difícil de cambiar es hacer que muchos otros paquetes de software dependan de él. Un paquete con muchas dependencias entrantes es muy estable porque requiere una gran cantidad de trabajo para reconciliar cualquier cambio con todos los paquetes dependientes.

- **Métricas de estabilidad**. Podemos calcular la estabilidad de un paquete usando un trío de métricas simples.
  - *Ca*: Acoplamiento aferente. El número de clases fuera del paquete que dependen de las clases dentro del paquete.
  - *Ce*: Acoplamiento eferente. El número de clases fuera del paquete de las que dependen las clases dentro del paquete.
  - *I*: Inestabilidad. I=Ce/(Ca+Ce) Esta es una métrica que tiene el rango: [0,1]

    Si no hay dependencias salientes, seré cero y el paquete será estable. Si no hay dependencias entrantes, seré una y el paquete es inestable.

- **El principio de abstracciones estables (SAP):** *Los paquetes estables deben ser paquetes abstractos.* El SAP es solo una reformulación del DIP. Establece que los paquetes de los que más se depende también deben ser los más abstractos.
  - **La métrica de abstracción**. Podemos derivar otro trío de métricas para ayudarnos a calcular la abstracción.
    - **Nc:** Número de clases en el paquete.
    - **Na**: Número de clases abstractas en el paquete.
    - **A**: Carácter abstracto.A=Na/Nc

    La métrica A tiene un rango de [0,1]. Un valor de cero significa que el paquete no contiene clases abstractas. Un valor de uno significa que el paquete no contiene nada más que clases abstractas.

  - **El gráfico I vs A**. El SAP ahora se puede reformular en términos de las métricas I y A: I debería aumentar a medida que A disminuye. Es decir, los paquetes concretos deben ser inestables mientras que los paquetes abstractos deben ser estables.
  - **Métricas de distancia**. Esto nos deja un conjunto más de métricas para examinar. Dados los valores A e I de cualquier paquete, nos gustaría saber qué tan lejos está ese paquete de la secuencia principal.
    - **D**: Distancia.D= (|A+I- 1|)/√2 . Esto oscila entre [0,~0,707].
    - **D'**: Distancia normalizada. D = |A + I - 1| . Esta métrica es mucho más conveniente que D ya que oscila entre [0,1]. Cero indica que el paquete está directamente en la secuencia principal. Uno indica que el paquete está lo más alejado posible de la secuencia principal.

Patrones de Diseño

Con el tiempo los desarrolladores se dieron cuenta que independientemente del dominio de aplicación, existían entre varios proyectos, problemas en común. A partir de esto, lo que se empezó a pensar fue como crear soluciones a esos problemas que fuesen fácilmente adaptables a cada uno de estos dominios de aplicación. La principal idea de esto, es la reutilización de código, así nacen los **patrones de diseño**.

Los patrones de diseño son ideas, conceptos, formas de organizar un conjunto de clases o un conjunto de métodos para favorecer la solución de un problema manteniendo la reusabilidad y la extensibilidad de esa solución.

Los patrones se aplican en situaciones especiales

Todos los patrones comparten ciertas características:

- Poseen un nombre que lo identifican.
- Para cada patrón vamos a tener una motivación, cuál es la razón de existir de ese patrón, cuál es problema que se está intentando solucionar a partir de ese patrón.
- Una estructura de clases.

Los patrones se organizan en 3 grupos:

- Los creacionales: nos ayudan en la creación de objetos.

- De comportamiento: nos permiten modificar el comportamiento, las responsabilidades de un conjunto de clases.
- De estructura: nos ayudan a organizar estructuras complejas a partir de clase u objetos más simples, estructuras que sean fácilmente extensibles y muy flexibles.

Los patrones de diseño proveen una forma efectiva de guía en el desarrollo de software. Los anti-patrones son un concepto relativamente nuevo en la ingeniería de software y una extensión natural a los patrones de diseño. Pretenden ayudar en el proceso de desarrollo, de forma similar a como lo hacen los patrones de diseño.

Los anti-patrones sin soluciones negativas que presentan más problemas que los que soluciona. ¿Por qué estudiar los anti-patrones? La idea no es tanto decir "*no hagas esto*" sino "*tal vez no sepas que haces esto… pero no funciona*". Permite reconocer problemas comunes y sus causas e implementar soluciones productivas. Proveen un vocabulario común para la identificación de problemas y sus soluciones.

Los anti-patrones determinan un proceso de escritura iterativo:

- Identificar el Anti-patrón.
- Definir la solución.
- Definir síntomas y consecuencias.
- Revisar y elaborar.

Los anti-patrones pueden agruparse en:

- **De desarrollo de software**: describen situaciones usualmente candidatas a la refactorización.
- **De la arquitectura del software**: se enfoca en la estructura general del sistema y sus componentes.
- **De administración del proyecto**: se enfocan en aspectos relacionados con la administración del proyecto, de las personas involucradas y de las comunicaciones humanas.

Patrones creacionales

Los patrones creacionales tienen que ver con darnos flexibilidad al momento de crear o generar instancias de una clase. Lo que nos va a evitar estos patrones es tener que hacer un llamado explícito a **new**. El uso del new no se pierde, solo que se realiza de manera implícita a partir de otro método. Procuran independizar el sistema de cómo sus objetos son *creados, compuestos y representados*.

Los patrones indican soluciones de diseño para **encapsular** el conocimiento acerca de las clases que el sistema usa y **oculta** como se crean instancias de estas clases.

Dentro de esta categoría existen distintos patrones:

- Abstract Fanctory: es un patrón que nos dará flexibilidad al crear distintos tipos de objetos, familias de objetos. Lo que sugiere este patrón es:
    1) Armar sobre esa familia de productos una jerarquía de herencia.
    2) Luego, aparecen las **Factories**, las cuales son las clases encargadas de crear las instancias.
    3) Las fábricas también poseen una jerarquía de herencia, iniciando con una fábrica abstracta que define la interfaz de las fábricas concretas. Voy a tener tantas fábricas concretas como familias de productos tenga.

  Participantes:
    - AbstractFactory: declara la interfaz.
    - ConcretFactory: implementa las operaciones.
    - AbstractProduct: declara una interfaz para un tipo de producto.
    - ConcretProduct: define un objeto producto que será creado por la correspondiente clase Factory concreta.
    - Cliente: usa solo interfaces declaradas por AbstractFactory y AbstractProduct.
- Prototipe: muchas veces crear objetos es muy costoso, es tan costoso que lo mejor es hacerlo la menor cantidad de veces posible, pero a su vez lo que queremos es tener muchas instancias. Este propone crear una sola instancia de ese objeto y luego generar clones de la misma. Usamos este patrón cuando:

- No es posible duplicar una instancia usando solo sus métodos públicos.
- Instancias de una clase pueden tener solo uno de muchas combinaciones de Estados.

Participantes:

- Prototype: declara una interfaz para la autoclonación.
- ConcretePrototype: implementa una operación para clonarse a sí mismo.
- Client: crea un nuevo objeto solicitándole al prototipo que se Clone a su mismo.

- Singleton: este patrón nos permite tener una única instancia de una clase en todo el sistema. ¿Cómo se logra esto? en la clase donde vamos a usar este patrón de diseño, se declara como **privado** el constructor (de esta manera no va a ser posible crear instancias de esa clase) y se crea un método **estático** que se va a encargar de controlar si ya se creó una instancia de esa clase, si no fue así creará una, en caso contrario devolverá la instancia creada anteriormente. Usamos este patrón cuando:
  - Queremos controlar el acceso a un recurso compartido.
  - No queremos tener dos instancias de diferentes tipos en un mismo sistema.
- Builder: se aplica en una situación en la que la construcción de un objeto es compleja. En particular, la situación en la que el constructor es complejo. Lo que realiza la clase builder es simplificar el proceso de crear objetos. Este método de simplificación consiste en iniciar la construcción de un objeto y solo indicarle qué parámetros nos interesa completar (a través de los métodos). Similar al abstract factory, el builder nos permite tener una clase abstracta y después tener builders concretos. Usamos este patrón cuando:
  - La construcción de un objeto requiere muchos parámetros que no siempre son instanciados.
  - La construcción de un objeto se hace mediante métodos en la clase del objeto que se quiere construir.

Una particularidad que tiene todo esto, es que muchas veces para ganar flexibilidad para escribir menos código el día de mañana, es necesario escribir más código el día de hoy.

Patrones estructurales

Los patrones estructurales son aquellos que nos permiten organizar, ensamblar objetos o clases en estructuras más grandes, siempre con una perspectiva de mantener la flexibilidad y ser eficiente en el uso de las clases y objetos.

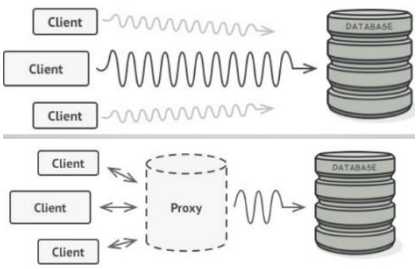
Existen distintos tipos de patrones estructurales:

- Adapter El adaptador es un objeto que implementa la interfaz abstracta para delegar al servidor. Cada método del adaptador simplemente traduce y luego delega. Usaremos este patrón cuando:
  - Deseamos usar una clase existente, pero su interfaz no es la que necesitamos.
  - Queremos crear una clase reutilizable que coopera con otras clases no relacionadas, probablemente con interfaces incompatibles.
  - Necesitamos usar varias subclases, pero no es práctico adaptar sus interfaces heredando de ellas, por lo que apelamos a un objeto adaptador.

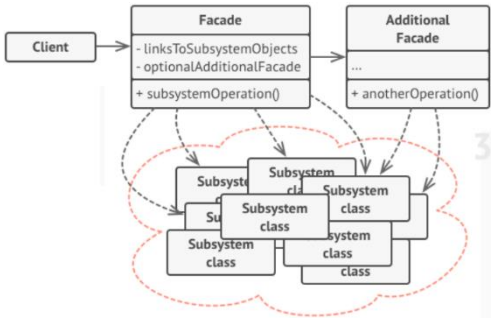
Participantes:

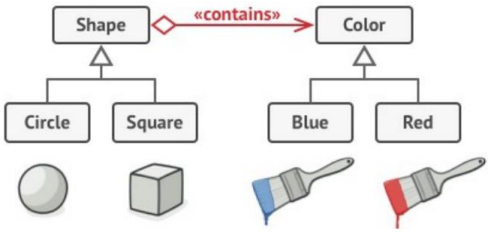
- Target: define la interfaz que el cliente usa.
- Adaptee: interfaz existente que necesita adaptarse.
- Adapter: adapta la interfaz Adaptee a la que necesita el cliente.

- Proxy: Esto se plantea con una interfaz la cual va a poseer los métodos que voy a utilizar y con la cual se va a comunicar todo mi programa, mi clase proxy implementa esa interfaz y, a diferencia del patrón adapter, proxy controla lo que realiza la tercera clase (en lugar de adaptar esa clase).



- Facade: Proveer una interfaz simplificada para acceder a una librería, framework o conjunto complejo de clases.
- Bridge: si se tiene una clase que posee más de una responsabilidad, lo que plantea este patrón es dividir la clase en dos o más (dependiendo las responsabilidades) y establecer un puente entre ellas. Separar una gran clase o conjunto de clases en dos jerarquías, **abstracción** e **implementación** las cuales pueden ser desarrolladas independiente mente.





- Composite: compone objetos en estructuras de árbol para representar jerarquías de relación "es parte de". Usaremos este patrón cuando:
    - Queremos representar jerarquías de objetos modelando la relación "es parte de".
    - Queremos que el cliente ignore la distinción entre objeto compuesto y objeto individual.
- Decorator: agrega responsabilidades a un objeto de manera dinámica. Provee una alternativa a la herencia cuando deseamos agregar funcionalidad. Usaremos este patrón cuando:
    - Queremos agregar responsabilidades a objetos individuales de manera dinámica y transparente, sin afectar otros objetos.
    - Deseamos implementar responsabilidades que pueden removerse.
    - Cuando la extensión utilizamos herencia es impracticable.

  Participantes:
    - Component: define la interfaz para los objetos que pueden tener responsabilidades agregadas dinámicamente.
    - Concrete imponente: define un objeto al cual se le puede agregar responsabilidades dinámicamente.
    - Decorator: mantiene una interfaz a un objeto Component y define la interfaz que conforma la interfaz de Component.
    - ConcreteDecorator: agrega responsabilidades al componente.

Patrones de Comportamiento

Los patrones de comportamiento se tratan con algoritmos y la asignación de responsabilidades entre objetos. Los patrones de comportamiento de clases utilizan herencia para distribuir el comportamiento entre las clases. Los patrones de comportamiento de objetos utilizan composición de objetos en lugar de herencia. Estos patrones son:

- Command: Encapsular el pedido (mensaje, solicitud) como un objeto, permitiendo parametrizar los clientes con diferentes pedidos, encolar o registrar los pedidos y proveer operaciones para deshacer pedidos previos. Usaremos este patrón cuando deseamos:
    - Parametrizar objetos para una acción a realizar.
    - Especificar, encolar y ejecutar pedidos en momentos diferentes.
    - Proveer la posibilidad de deshacer acciones.
    - Proveer registros de auditoría y salvaguarda.
    - Estructurar un sistema en función de operaciones de alto nivel construidas en base a operaciones primitivas.

  Participantes:
    - Command: declara una interfaz para ejecutar operaciones.
    - ConcreteCommand: define el vínculo entre el objeto Receiver y una acción. Implementa Execute() invocando las operaciones correspondientes sobre el Receiver.

    - Client: crea un objeto ConcreteCommand y setea el receptor.
    - Invoker: solicita al comando realizar su tarea.
    - Receiver: conoce cómo realizar operaciones asociadas al llevar a cabo un pedido. Cualquier clase puede actuar como Receiver
- Observer: Define una dependencia entre objetos de uno-amuchos de forma tal que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados acordemente. Usaremos este patrón cuando:
    - Cuando una abstracción tiene dos aspectos, uno independiente del otro.
    - Cuando un cambio a un objeto requiere cambios en otros, y no sabemos cuántos objetos necesitan ser cambiados.
    - Cuando un objeto debería notificar a otros objetos sin realizar suposiciones de quiénes son esos objetos (evitar el acoplamiento)
- State: es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiará su clase.
- Strategy: es un patrón de diseño de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.
- Visitor: es un patrón de diseño de comportamiento que te permite separar algoritmos de los objetos sobre los que operan.

Diseño Podrido

El diseño de muchas aplicaciones de software comienza como una imagen vital en la mente de sus diseñadores. Tiene una belleza simple que hace que los diseñadores e implementadores anhelen verlo funcionar.
Pero entonces algo comienza a suceder. El software comienza a pudrirse. El programa se convierte en una masa enconada de código que los desarrolladores encuentran cada vez más difícil de mantener.

Hay cuatro síntomas principales que nos dicen que nuestros diseños se están pudriendo ortogonalmente, pero están relacionados entre sí de maneras que se volverán obvias. Ellos son:

- La **rigidez** es la tendencia del software a ser difícil de cambiar. Cada cambio provoca una cascada de cambios posteriores en los módulos dependientes.
- Estrechamente relacionado con la rigidez está la **fragilidad**. La fragilidad es la tendencia del software a romperse en muchos lugares cada vez que se cambia. A medida que la fragilidad empeora, la probabilidad de rotura aumenta con el tiempo, acercándose asintóticamente.
- La **inmovilidad** es la incapacidad de reutilizar software de otros proyectos o de partes del mismo proyecto.
- La **viscosidad** se presenta en dos formas: la viscosidad del diseño y la viscosidad del entorno. Cuando se enfrentan a un cambio, los ingenieros suelen encontrar más de una forma de realizar el cambio. Algunas de las formas conservan el diseño, otras no. Cuando los métodos de preservación del diseño son más difíciles de emplear que los hacks, entonces la viscosidad del diseño es alta. La viscosidad del entorno surge cuando el entorno de desarrollo es lento e ineficiente.

La causa inmediata de la degradación del diseño es bien conocida. Los requisitos han ido cambiando de maneras que el diseño inicial no anticipó. Entonces, aunque el cambio en el diseño funciona, de alguna manera viola el diseño original. La mayoría de nosotros nos damos cuenta de que el documento de requisitos es el documento más volátil del proyecto.

¿Qué tipo de cambios hacen que los diseños se pudran? Cambios que introducen dependencias nuevas y no planificadas. Lo que se está degradando es la arquitectura de dependencia y, con ella, la capacidad del software para mantenerse.

Refactory

Se trata de acomodar la parte interna del sistema

Ese es el proceso a través del cual nosotros cambiamos como un sistema o parte de este está implementado sin modificar su funcionalidad. ¿Cuándo hacer refactoring? Cuando tomamos decisiones en el pasado sobre como acomodar el programa, las clases, etc. Y con el transcurso del desarrollo del proyecto nos arrepentimos de esas decisiones.

Hay varias razones para realizar refactoring:

- **Código duplicado**.
- Hace varios años, un investigador definió una métrica que se conoce como "**Complejidad Ciclomática**". Esta métrica nos indica que tan complejo es un código. Complejidad ciclomática es una forma de evaluar donde yo puedo llegar a hacer refactoring.
- "**Malos Olores**" o "**Smell**" en el código, los cuales son situaciones que en un futuro va a forzar el realizar un refactoring.

Se le dice código complejo a un código que posee:

- While
- For
- If
- Comparaciones

Existen varias herramientas dedicadas a verificar si en el código se encuentran las anteriores razones mencionadas.

3 capas

Lo que propone el modelo es estructurar u organizar toda la aplicación en 3 capas o 3 niveles. Que normalmente se identifican con:

- Vista: todo lo que ve el usuario, la interfaz.
- Lógica: se encarga de resolver la razón por la cual existe la aplicación.
- Datos: se encarga de manejar, guardar y recuperar los datos.

Estas capas lo que representan son organizaciones. Es una forma de pensar la solución a un problema pensándola separando responsabilidades. Lo más importante es que exista una comunicación muy clara y concreta entre las capas. De esta manera, lo que busco es evitar mezclar las capas.

Diseño por contrato

Una relación entre objetos en la cual un objeto ofrece servicios y otro utiliza sus servicios se conoce como relación **cliente/servidor**. Esta relación es la base del **diseño por contrato**.

Una situación que suele ocurrir en este tipo de relación es que el servidor tiene condiciones para ofrecer un servicio, la cual se llama **precondición**. A su vez puede asegurar cosas al finalizar el llamado, la cual se llama **postcondición**. Y dentro de lo que es diseño por contrato, toda operación puede tener una precondición y una postcondición (puede ser que tenga una, ambas o ninguna).

Básicamente, la precondición y la postcondición son condiciones booleanas. Pueden ser cualquier condición booleana (simple o compleja).

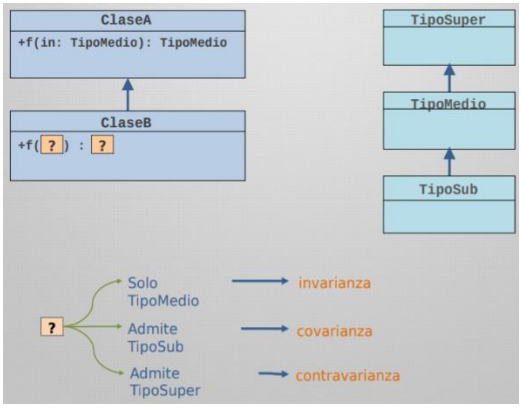
Las precondiciones y postcondiciones han ido evolucionando con el tiempo y en la actualidad pueden ser hasta documentación. Si se tiene una herramienta que computa las precondiciones y postcondiciones, nos permite simplificar el código y por lo tanto reduce las probabilidades de error.

Aspectj: herramienta que permite computar las precondiciones y las postcondiciones

Las precondiciones y postcondiciones surgen en la etapa de análisis del problema. Si no se posee esa herramienta que compute las condiciones, en todo el ciclo del software estoy dependiendo de que los seres humanos recuerden tener que agregar las condiciones.

De lo que se trata el diseño por contrato es de asegurar o maximizar las posibilidades de que nuestro programa salga bien. Esto se hace dándole mucha importancia a la especificación y trasladando las condiciones que nos da la especificación a nuestro programa.

El diseño por contrato nos introduce el concepto de **invariantes**, estas son condiciones booleanas que valen para toda la existencia de cada objeto/instancia de una clase.



El cliente tiene que ser el responsable de que se cumpla la precondición de una función y el servidor tiene que asegurarse de que todo lo que yo ponga en la precondición tiene que ser accesible desde afuera. Al momento de ejecutarse una función de la clase servidor, es muy importante que se cumpla antes de iniciar la precondición y la invariante y al finalizar se cumpla la postcondición y la invariante, puede pasar que durante la ejecución de la función no se satisfaga la invariante.

Las **aserciones** son el uso de condiciones booleanas para satisfacer el correcto funcionamiento de las funciones y de los objetos. Es algo que nos sirve para mejorar y asegurar la calidad y la correctitud del software.

Al momento de hablar de herencia el diseño por contrato puede volverse un poco complicado y pueden darse situaciones difíciles de explicar. Cada clase va a poseer su propio contrato que va a estar vinculado con el contrato de la clase padre. Al nivel de las invariantes, cada clase va a poseer sus propias invariantes.

Al hablar del principio de sustitución, al sustituir una clase por uno de sus descendientes, este debe cumplir las invariantes de el mismo y de su clase padre (si esta clase no es la original por la que se sustituyó, también se deben agregar su invariante y las de su padre hasta llegar hasta la clase original de la sustitución).

Dentro de las condiciones booleanas pasa que unas condiciones son más fuentes que otras. Se dice que una condición booleana {A} es más fuerte que una condición {X} cuando {A} implica {X} (esto quiere decir que si {A} es verdadero entonces {X} va a ser verdadera). Intuitivamente, si {A} es más fuerte que {X}, entonces {X} es más débil que {A}.

En la herencia, al momento de redefinir un método, existen varias reglas que se aplican a las pre y post condiciones:

- Como el principio de sustitución tiene que seguir siendo válido, todo lo que es válido para un método en la clase padre tiene que ser válido para el método de la clase hija.
    - Hablando de precondiciones, el filtro de la clase padre es más chico que el filtro de la clase hija (los valores de la precondición de la clase padre están incluidos dentro de los valores de la precondición de la clase hija).
    - Hablando de postcondiciones, el filtro de la clase padre es más grande que el filtro de la clase hija (los valores de la postcondición de la clase hija están incluidos en los valores de la postcondición de la clase padre).

Excepciones

Excepción ≠ Falla

Existen situaciones en las que la ejecución de nuestro programa se encuentra con situaciones que no puede resolver, una situación excepcional. Una **excepción** es un evento anormal producido durante la ejecución de un programa y que puede provocar que una operación falle. Vamos a llamar falla a la interrupción de un programa, en cambio una excepción no necesariamente implica una interrupción. Tener una falla, implica una excepción, pero una excepción no implica una falla.

¿Qué es eso de manejar una excepción? en ejecución de un programa son muchos objetos interactuando entre sí y eso produce una **cadena de llamados**. En algún punto de esa cadena de llamados puede surgir una excepción, en ese punto es donde hay que tomar la primera decisión que hago con esa excepción:

- La manejo con un *try/catch.* puede ocurrir que, al intentar manejar la excepción, falle y delegue el manejo de la excepción.
- Delega el manejo de esa excepción a quien me llamo.

Las excepciones pueden ocurrir por muchas razones y estas pueden ser manejadas o no, eso depende de cada tipo de excepción.

En Java, las clases que heredan de “Error” hacen referencia a errores propios del sistema. Que nuestro programa no debe manejar.

Java nos da una estructura de clases asociadas a las excepciones. En java, partimos de la clase “Object” y de esta clase extendemos una clase llamada “Throwable”. De esta clase heredan 2 clases que nos interesan, “Exception” y “Error”.

Cuando hablamos de la clase “Exception”, también habla de una gran jerarquía de herencia. Estos son errores que nuestro programa debe manejar.

Con esto hablamos de 2 tipos de excepciones en java:

- Cheched Exception: son aquellas excepciones que cuando el compilador detecta que hay una situación que puede generar una Cheches Exception, el compilador nos va a obligar a hacer algo con eso. Cuando el compilador nos avisa sobre una excepción, nosotros debemos:
  - Manejarlas (con try/catch).
  - Lanzarlas (con Throws).
- Uncheched Exception: son aquellas excepciones en las que el compilador no nos obliga a hacer nada.

Como lanzar una excepción:

Public parámetro_salida f(Parámetros) **throws** tipo_Exception

{dentro del código se debe agregar una línea de código con lo siguiente: throw new tipo_Exception ()}

Al momento de utilizar una operación que lanza una excepción, esa excepción es una cheched exception que detecta el compilador y la clase cliente debe manejar o propagar.

La estructura del try/catch es la siguiente:

- Bloque try: encierra todas las líneas de código las cuales yo quiero mirar por lanzamiento de excepciones. Las variables definidas dentro del try son locales al try.
- Bloque catch: captura la excepción y la maneja o lanza una nueva.
- Bloque finally: líneas de código que pase lo que pase en el bloque try, se van a ejecutar.

Puede haber múltiples bloques catch