Análisis y Diseño de Sistemas

Tema: Proceso RUP Agil
Diseño de Software para el reuso

Mg. Ing. Urciuolo A.

UNTDF, 2015

Agenda

- Enfasis en la segunda iteración: Diseño para el reuso.
- Reuso de Software
- Soporte de Reuso en OO
- Uso de Patrones
  - Conceptos generales
  - Problemas de diseño
  - Patrones GoF
- Diseño ágil y reuso

RUP Agil. Dónde estamos???

Fase de Elaboración. Iteración II

Diseño para el Reuso

Comenzando la Iteración II (Fase de Elab.)

- Cuando finaliza la Iteración 1:
  - ✓ Se han realizado las pruebas de todo el software (unidad, aceptación, usabilidad, etc.)
  - ✓ Los usuarios han evaluado la versión parcial del sistema, y se ha obtenido su visión para realizar cambios y mejoras
  - ✓ El sistema ha sido integrado y estabilizado como un producto interno de base

Comenzando la Iteración II (Fase de Elab.)

- Actividades que se deben realizar al finalizar la Iteración 1 y comenzar la 2 (aunque se han omitido en el curso):
  - ✓ Reunión de Planificación de la Iteración
  - ✓ Diseño de la Interfaz de usuario y análisis de Usabilidad.
  - ✓ Modelado e implementación de la Base de Datos.
  - ✓ Otro Workshop de Requerimientos para especificar nuevos Casos de Uso en formato completo (hasta un 80%), aunque muchos de ellos sean implementados en posteriores iteraciones.

El Enfasis en la Iteración II (Fase de Elab.)

- Durante el estudio de la Fase de Inicio y la Primera Iteración de la Fase de Elaboración se puso el énfasis en habilidades de análisis y diseño en orden a definir los *pasos para construir Sistemas OO.*

- ***Para el estudio de esta iteración se enfatizará:***
  - Diseño esencial de Objetos para el Reuso de software
  - El uso de Patrones para crear diseños sólidos
  - Aplicación de UML para visualizar los modelos

- **Todas las actividades correspondientes a las demás disciplinas (requerimientos, dominio, implementación, etc) se realizan en esta iteración, si bien en el curso, el énfasis se pone en el Diseño**

Reuso de Software

- En la mayoría de las disciplinas de Ingeniería, los sistemas se diseñan a partir de la composición de componentes existentes utilizados en otros sistemas.
- El proceso de diseño se basa en la reutilización de sistemas o componentes existentes
- La Ingeniería de SW se ha enfocado más en el desarrollo original, pero en la actualidad se reconoce que para alcanzar mejor SW, más rápido y a menor precio, se necesita adoptar un *enfoque de diseño basado en el reuso sistemático de SW.*

Unidades de Software reusables

- ***Reutilización de sistemas de aplicaciones.*** La totalidad de un sistema de aplicaciones puede ser reutilizada incorporándolo sin ningún cambio en otros sistemas (Reuso de COTS) o desarrollando familias de aplicaciones que tienen una arquitectura común pero que son adaptadas a clientes particulares.
- ***Reutilización de componentes.*** La reutilización de componentes de una aplicación varía en tamaño desde subsistemas hasta objetos simples.
- ***Reutilización de objetos y funciones.*** Pueden reutilizarse piezas de software que implementan una única función, como por ejemplo una función matemática o una clase de objetos. (Están disponibles muchas librerías de funciones y clases para diferentes tipos de aplicaciones y plataformas de desarrollo).

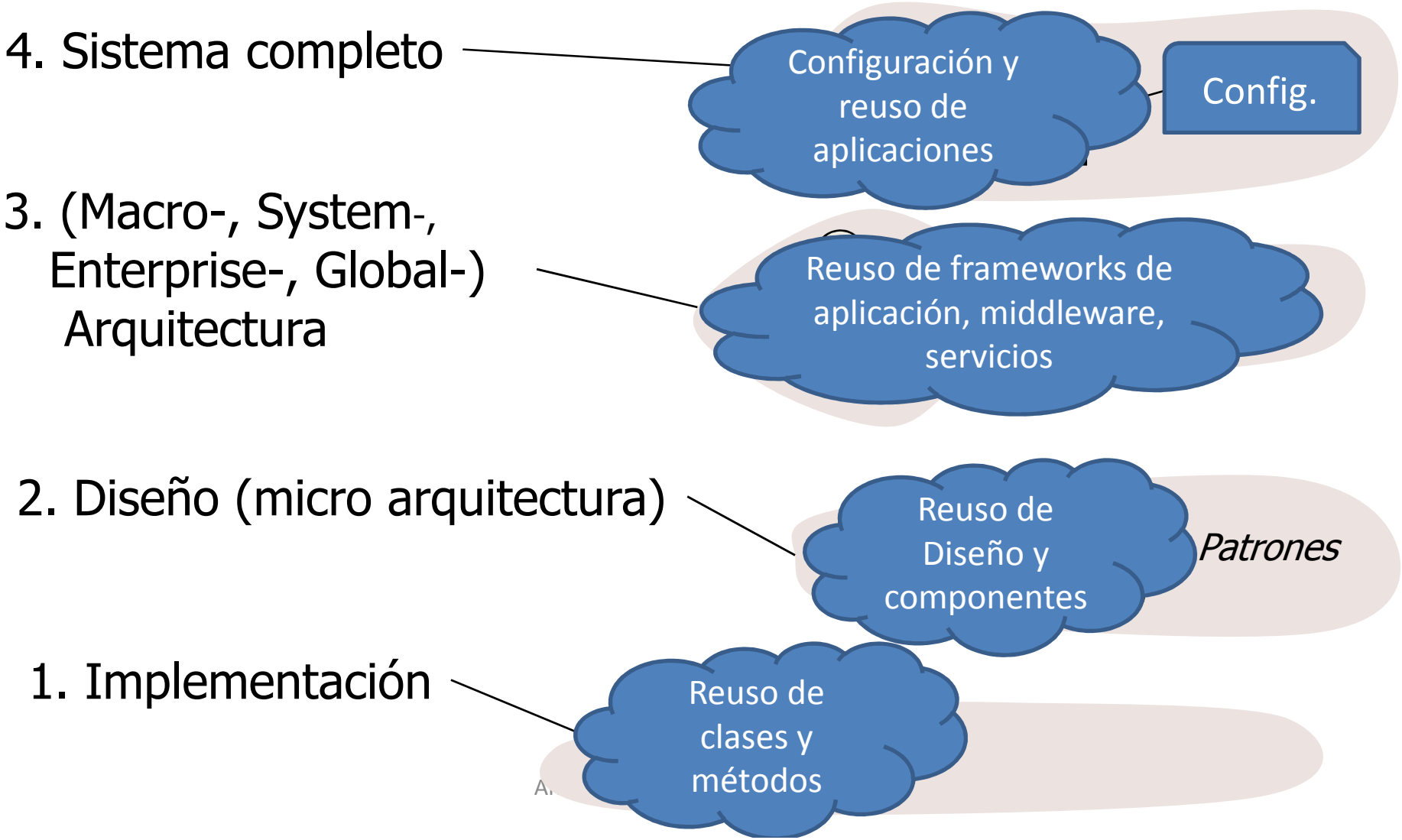
Reuso de conceptos

- Cuando se reusan programas o componentes ejecutables, se deben seguir las decisiones de diseño de quien los hizo y a veces el coste de modificarlos para una nueva situación resulta elevado.
- Esto limita las oportunidades para reuso.
- Una forma más abstracta de reuso es el ***reuso de conceptos,*** en la cual, en lugar de reutilizar un componente, la entidad reutilizada es más abstracta (no incluye detalles de implementación) y se diseña para ser configurada y adaptada a una variedad de situaciones.
- Los dos enfoques principales de reuso de conceptos son:
  - Patrones de diseño
  - Programación generativa

El campo del reuso

Fuente: Sommerville, 2007

Las 4 capas del reuso



Reuso Conceptual: Enfoques de soporte de reuso a nivel arquitectural

- Patrones arquitecturales
- Frameworks de aplicación
- Legacy system wrapping
- Service-oriented systems

Reuso conceptual: Enfoques de soporte de reuso a nivel de diseño de Objetos

- Diseño OO
- Patrones de diseño
- Model-driven engineering
- Aspect-oriented software development
- Component-based development

Recordar que el Diseño de Software abarca:

- **Diseño del sistema (Arquitectura)**
  - Transformación de un modelo de análisis (el qué) en un modelo de diseño (el cómo).
  - Durante el diseño del sistema, el mismo se descompone en subsistemas menores.
  - Se toman otras decisiones arquitecturales (distribución, concurrencia, etc.)
- **Diseño de Objetos (o diseño detallado)**
  - Transformación del modelo conceptual de objetos del análisis en un modelo que sirva de base para la implementación
  - Descripción de clases, estructuras de datos y especificación de interfaces de clases del sistema

Reuso a nivel de diseño de Objetos

- **Mezcla de buenas prácticas de diseño**
- **Algunos paradigmas pueden ayudar:**
  - Diseño OO
  - Patrones de Diseño
  - Model-driven engineering
  - Desarrollo de software orientado a Aspectos
  - Desarrollo basado en componentes

Diseño de Objetos

- Nos ocupamos del Diseño de Objetos
- Previamente:
  - Durante el análisis se describe la funcionalidad del sistema
  - Durante el diseño de sistemas, se describe el mismo en términos de su arquitectura, tal como la descomposición en subsistemas y se definen las plataformas de hardware/software sobre las cuales construir el sistema.
  - Durante el diseño de Objetos se reduce la brecha entre objetos del dominio y de la solución, se identifican nuevos objetos de la solución y se refinan los existentes.

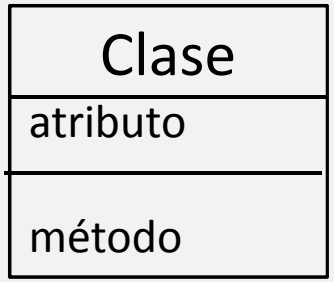
Diseño de Objetos

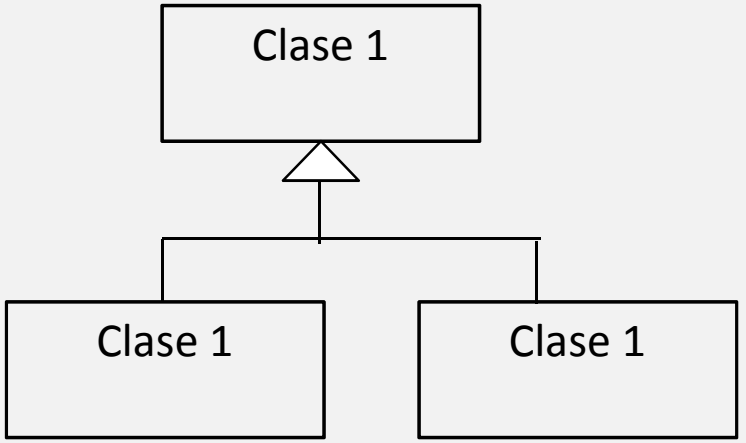
- Nos ocupamos del Diseño de Objetos
- Incluye:

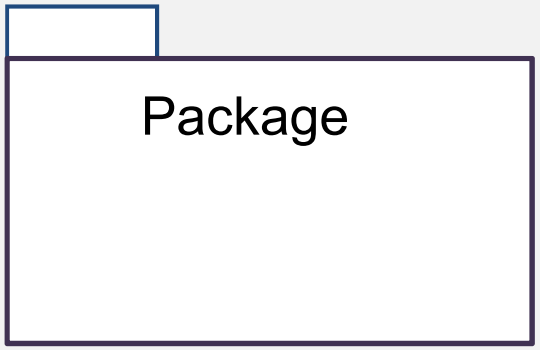
- **Reuso**, durante el cual identificamos componentes, y patrones de diseño para hacer uso de soluciones existentes
- **Especificación de servicios y asociaciones** durante la cual describimos en forma precisa las interfaces de cada clase y las asociaciones entre las mismas
- **Reestructuración del modelo de objetos**, durante el cual transformamos el modelo de diseño de objetos para mejorar su comprensibilidad y extensibilidad
- **Optimización del modelo de objetos** durante el cual lo transformamos para manejar criterios de performance

OO. Soporte de Reuso.

- **Principales características OO para soporte de reuso**
  - **Encapsulación, modularización, herencia**





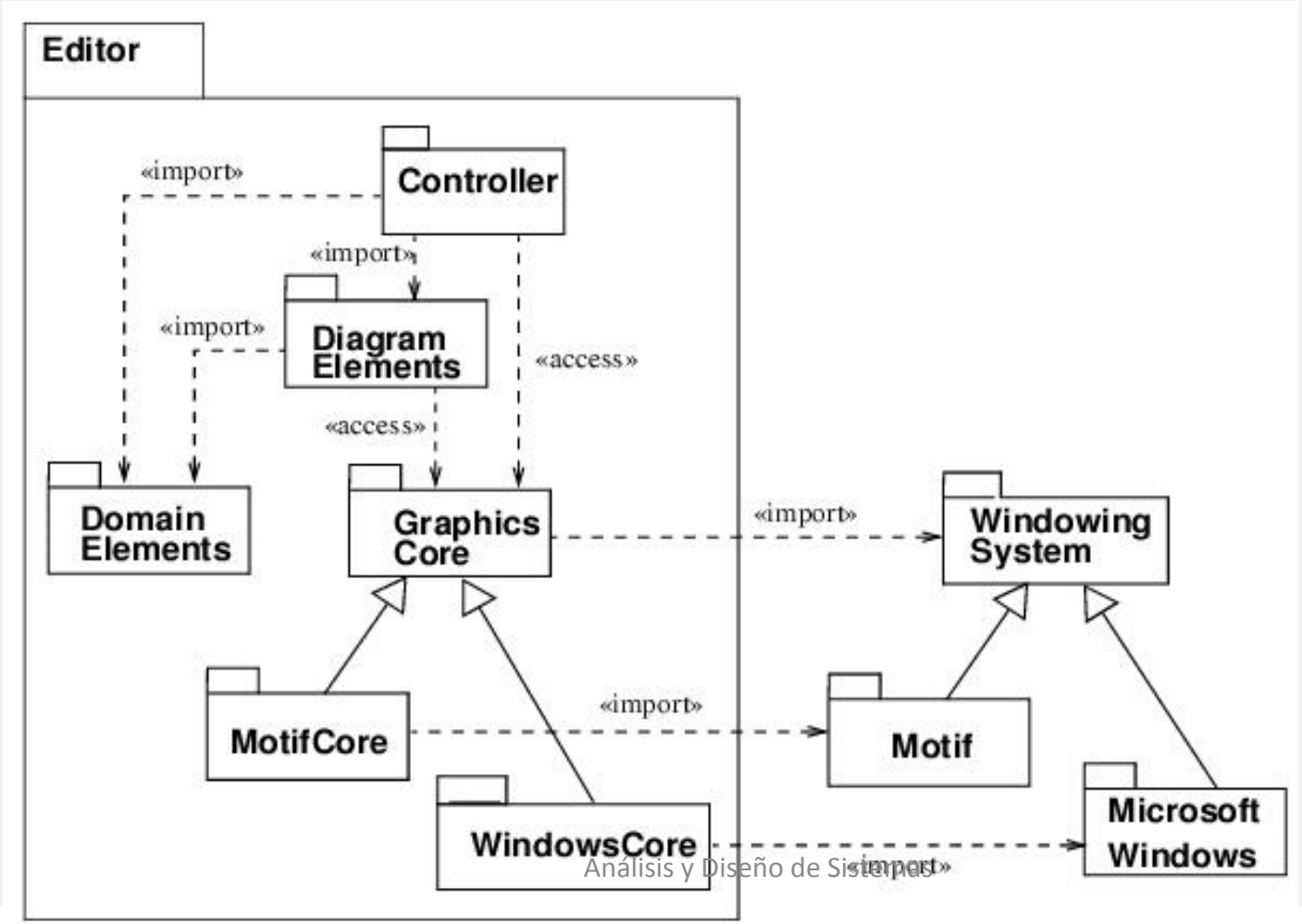


OO. Soporte de Reuso.

- **Encapsulación y Ocultamiento de información**
  - Se expone sólo lo suficiente de un módulo para permitir que otros módulos hagan uso de él.
  - Se puede sintácticamente impedir el paso a detalles de implementación, dando mayor flexibilidad y mantenibilidad al sistema.
  - El ocultamiento de detalles en una cápsula brinda un mayor potencial para reuso.
  - Cada ítem en el sistema puede cambiar independientemente, sin impactar otros módulos.

OO. Soporte de Reuso.

- **Interfaces, packages:** mecanismos básicos que apuntan a la modularización (y de esa forma, al reuso)



OO. Soporte de reuso

- **Principales características OO para soporte de reuso**
  - **Overriding, Sobrecarga y Polimorfismo: son los mecanismos concretos para el reuso basado en herencia.**
  - **Overriding**, se escribe en una subclase un método con el mismo nombre y parámetros que en la superclase → *se reusa el resto de la superclase*
  - **Sobrecarga** se escriben múltiples métodos con el mismo nombre, pero con diferentes lista de parámetros → *aparenta que el objeto tiene mayor probabilidad de reuso*
  - **Polimorfismo:** Objetos de varios tipos definen una interface común para usuarios. → los usuarios pueden compartir el uso, aunque en tiempo de ejecución se pueden enlazar instancias de diferentes tipos

OO. Soporte de reuso

```
public class Animal {
    public static void hide() {
        System.out.format("Hide animal."); }
    public void override() {
        System.out.format("Override Animal."); }
}
```

```
public class Cat extends Animal {
    public static void hide() {
        System.out.format("Hide Cat."); }
    public void override() {
        System.out.format("Override Cat."); }
}
```

```
public static void main(…) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        //myAnimal.hide();  //Mal estilo
        Animal.hide();        //Mejor
        myAnimal.override();
    }
}
```

OO. Soporte de reuso

- La clase Cat sobreescribe (overrides) el método de instancia en Animal llamado *override* y oculta el método de clase en Animal llamado *hide*
- Para métodos de clase, el sistema run-time invoca al método definido en el tipo compile-time de la referencia.
- Para métodos de instancia, el sistema run-time invoca al método definido en el tipo runtime de la referencia.
  - El método hide en Animal.
  - El método override en Cat.

Principios que orientan el reuso en OO

- **Responsabilidad única – Separación de cuestiones**
  - Una clase debe tener una única razón para cambiar
- **Principio Open/close**
  - Las entidades de Software deben estar abiertas para extensión pero cerradas a modificaciones.
- **Sustitución (Liskov)**
  - Si un programa está usando una clase base, luego la referencia a la clase base puede ser reemplazada con una clase derivada sin cambiar o afectar la funcionalidad general.
- **Inversión de Dependencia**
  - Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.
  - Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones.
- **Segregación de Interface**
  - Los clientes no deben ser forzados a depender de interfaces implementadas que no usan.

Responden a Problemas en el diseño

- **Rigidez:** El sistema es difícil de cambiar, porque cada cambio fuerza muchos otros en distintas partes del sistema.
- **Fragilidad:** Los cambios causan que el sistema falle en lugares que no están conceptualmente relacionados con la parte que cambió.
- **Inmovilidad:** Es difícil separar el sistema en componentes que puedan ser reusados en otros sistemas.
- **Viscosidad:** Aparece en proyectos en los cuales el buen diseño del SW es difícil de preservar ante cambios realizados
- **Complejidad innecesaria:** Contiene elementos que no son frecuentemente utilizados
- **Repetición innecesaria:** El diseño contiene estructuras repetitivas que podrían ser unificadas bajo una abstracción simple
- **Opacidad:** El diseño es difícil de leer y comprender, no expresa claramente su meta.

Principios de Diseño OO: SOLID

- SOLID es un acrónimo para 5 principios de diseño de objetos,
- Se utiliza en base a los "primeros cinco principios" de diseño y programación enunciados por Robert Martin en el año 2000.
- Estos principios se aplican todos juntos, y ayudan a que los sistemas sean más mantenibles y fáciles de extender en el tiempo.
- Se pueden aplicar los principios SOLID en código existente como guía para realizar refactorización y durante la codificación de sistemas nuevos.

Patrones
Conceptos y Generalidades

Soporte de Reuso. Patrones.

- Un Patrón de diseño es una solución reusable a un problema recurrente. Brindan una forma de reusar conocimiento abstracto acerca de un problema y su solución.
- Un patrón es la descripción del problema y la esencia de su solución.
- Debe ser lo suficientemente abstracto como para reusarse en diferentes ambientes
- Los patrones se apoyan en conceptos de diseño OO tales como polimorfismo, delegación, etc.

- ***Los patrones son formas de describir las mejores prácticas, buenos diseños, y encapsulan la experiencia de tal forma que es posible para otros el reutilizar dicha experiencia.***

Patrones

- El uso de patrones tiene sus orígenes en el trabajo de Christopher Alexander, un arquitecto, quien escribió ampliamente sobre el tema (relacionado con planeamiento urbano y arquitectura de edificios).

- Concepto del término patrón (Alexander et al, 1977):

  "Cada patrón describe un problema que ocurre una y otra vez en nuestro ambiente y luego describe la esencia de la solución a ese problema en tal forma, que se puede utilizar esa solución un millón de veces más, sin hacerlo en la misma forma dos veces".

- Las ideas de Alexander dieron origen a diversos trabajos de investigación sobre el tema, aplicando los patrones al desarrollo de software

Tipos de Patrones

- ***Patrones de código***
  - Nivel de lenguaje de programación
- ***Patrones conceptuales***
  - Nivel de Análisis
- ***Patrones arquitecturales***
  - Expresan la organización estructural global fundamental o esquema, de un sistema de Software.
- ***Patrones de diseño***
  - Descripciones de objetos y clases comunicantes que son personalizados para resolver un problema general de diseño, en un contexto particular (Gamma y otros, 1995)
- ***Patrones de Proceso***
- ***Nos concentramos en Patrones de diseño y arquitectura***

Ejemplos de Patrones

- **Patrones arquitecturales**
  - Layers (Capas)
  - Pipes and Filters (Tuberías y filtros)
  - Cliente/Servidor
  - MVC (Model- View-Controller)

- **Patrones de diseño**
  - Proxy
  - Factory Method
  - Adapter
  - Composite
  - Bridge

Patrones de diseño y arquitecturales

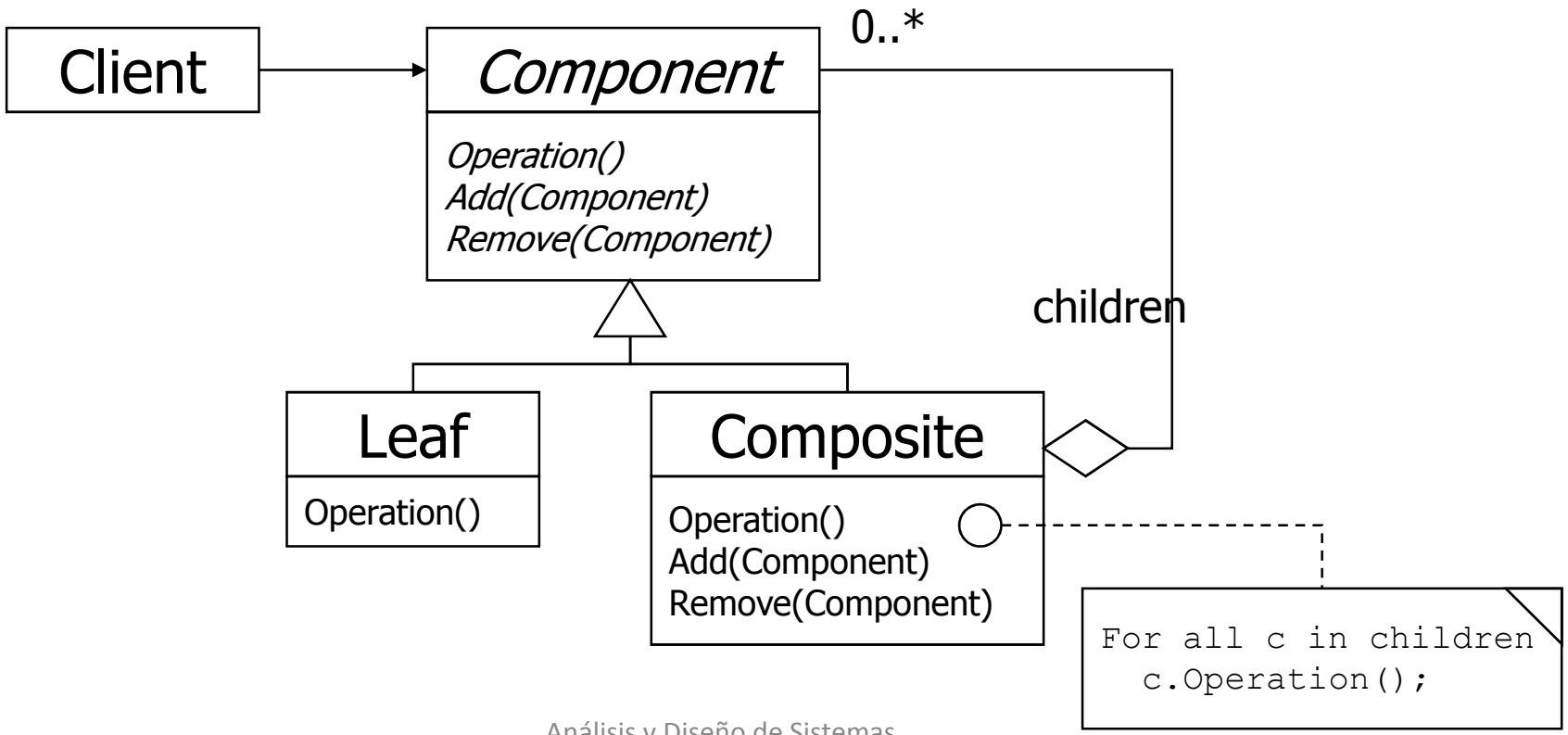
- **Los patrones de diseño describen soluciones a problemas de diseño recurrentes**.
  - Un patrón de diseño nombra, abstrae e identifica los aspectos claves de una estructura de diseño común que lo hacen útil para crear un diseño reusable.
  - Un patrón de diseño identifica: clases, roles, colaboraciones, y la distribución de responsabilidades.
- **Un patrón arquitectural expresa la organización estructural global fundamental o esquema, de un sistema de Software.**
  - Provee un conjunto de subsistemas predefinidos, especifica sus responsabilidades e incluye reglas y guías para organizar las relaciones entre los mismos (Buschmann *et al*, 1996).
- **La distinción es con respecto a la escala.**

  Los patrones de diseño tratan sobre las clases y sus interacciones

  Los patrones arquitectónicos tratan sobre subsistemas que son conjuntos de clases con una misión particular

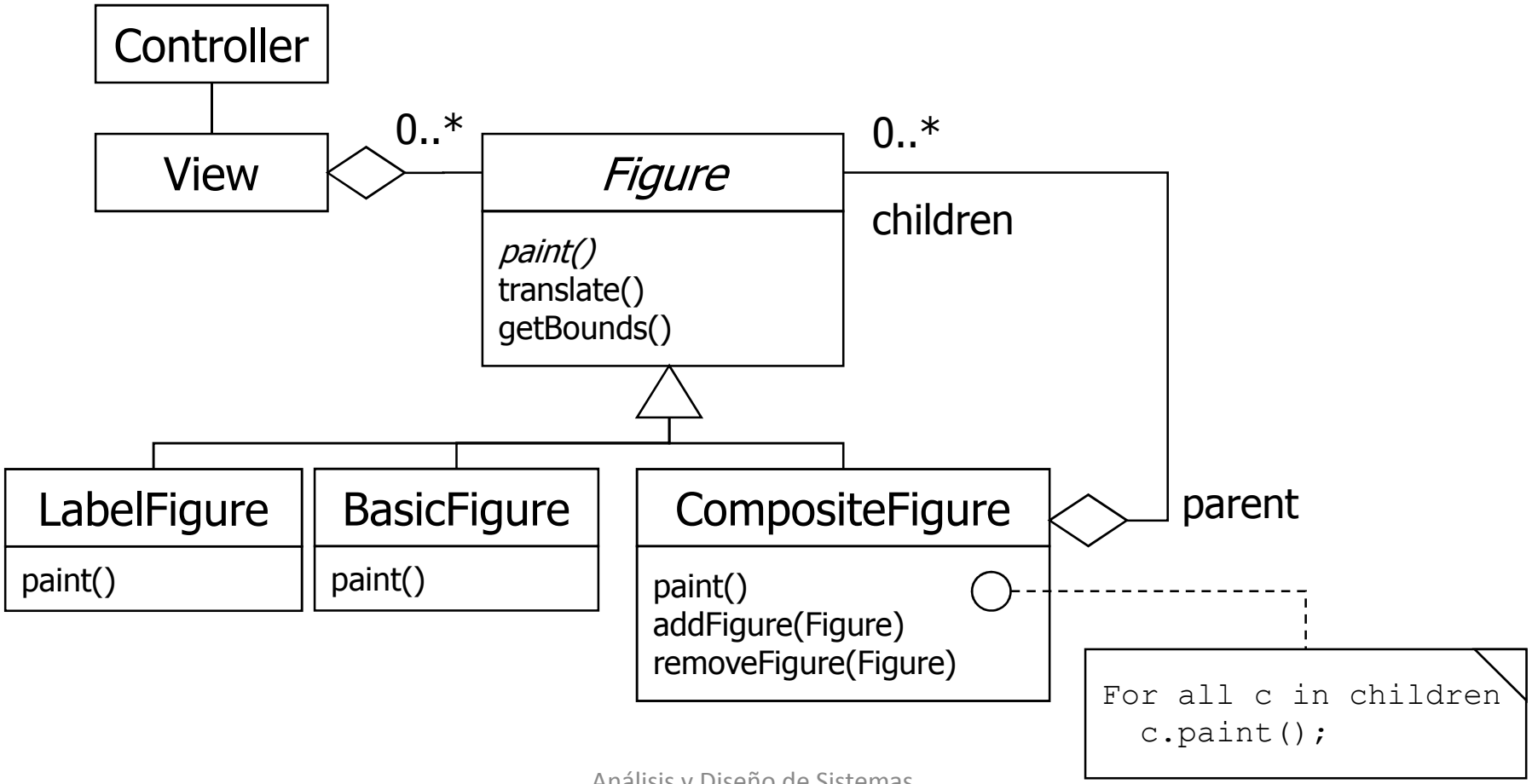
Ejemplo Patrón de diseño:Composite

- Compone objetos en estructuras de árbol para representar jerarquías todo-parte
- Permite que los clientes traten de manera uniforme a objetos individuales y compuestos
- Facilita agregar nuevos tipos de componentes



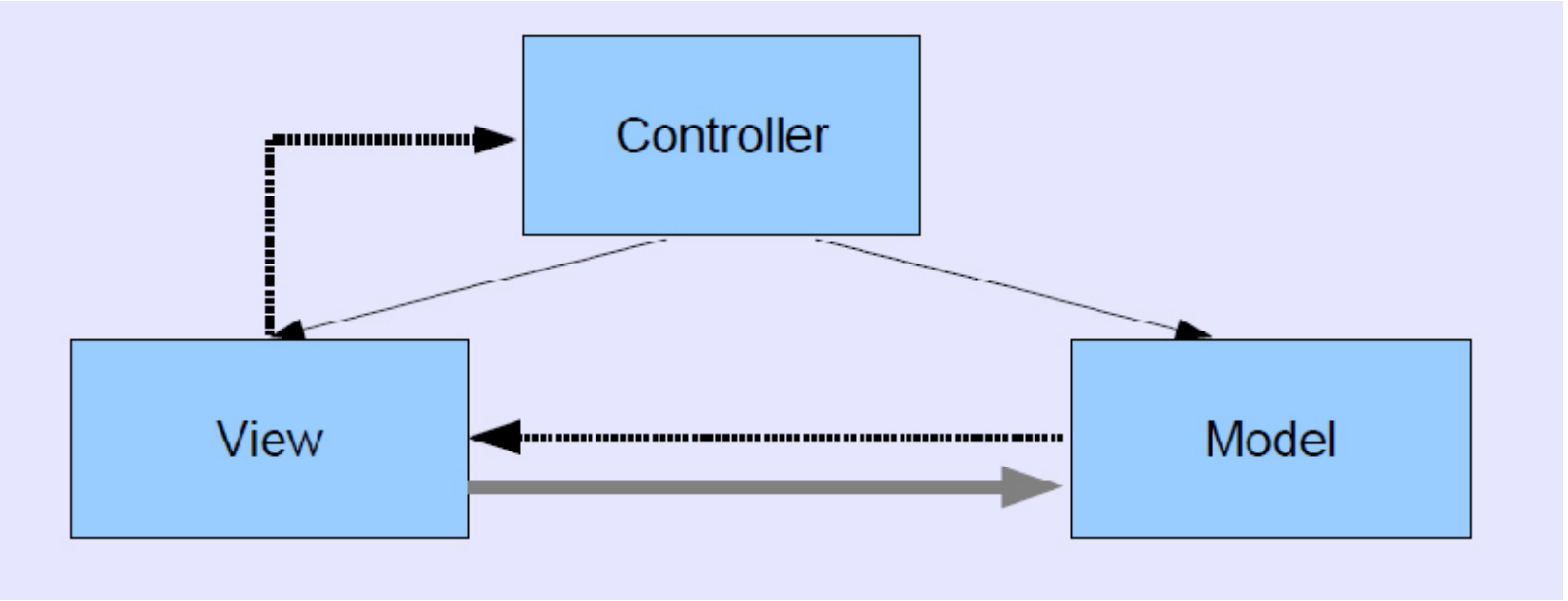
Ejemplo. Patrón de Diseño Composite

- Ejemplo: Figuras en un toolkit de gráficos
- La Clase Figura representa tanto primitivas como sus contenedores



Ejemplo. Patrón Arquitectural MVC

- Ejemplo: *MVC* es un patrón de arquitectura de software que separa el modelo del dominio y la lógica del negocio de su presentación
- MVC se basa fundamentalmente en el uso del patrón de diseño *Observer*



Ejemplo. Patrón Arquitectural MVC

- El modelo contiene las clases subyacentes para representar la lógica de negocio, cuyas instancias (objetos) serán vistas y manipuladas y contiene los datos.

- La vista contiene objetos usados para presentar la apariencia de los datos del modelo en la interfaz de usuario y los controles con los cuales el usuario puede interactuar.

- El controlador contiene los objetos que controlan y manejan la interacción del usuario con la vista y el modelo.
  - El patrón de diseño **Observer** es usado normalmente para separar el modelo de la vista.

Ejemplo. Patrón Arquitectural MVC

Ventajas de MVC:

- Claridad de diseño
- Incrementa la reusabilidad desacoplando parcialmente la presentación de los datos, la representación de los datos y las operaciones de aplicación.
- Permite mútiples vistas simultáneas.
- Facilita mantenimiento, extensibilidad, flexibilidad, y encapsulación, desacoplando capas de software

Patrones de Diseño. Características

- Describen un problema de diseño recurrente y una solución.
- Cada patrón nombra, explica, evalúa un diseño recurrente en sistemas OO.
- Elementos principales:
  - **Nombre**
  - **Problema**
  - **Solución:** Descripción abstracta
  - **Consecuencias**

Origen de los Patrones de Diseño

- Eric Gamma, Richard Helm, Ralph Johnson y John Vlissides publicaron el primer catálogo de patrones en el ámbito del software en 1994:
  - "Design Patterns: Elements of Reusable Object-Oriented Software, también conocido como GoF" ("Gang of Four")
  - Fue la primera vez que se documentó el conocimiento que usaban los expertos para resolver problemas de diseño
  - Se inspiraron en el trabajo de Christopher Alexander
  - Desde entonces, se han publicado muchos otros catálogos

Patrones. A qué ayudan en el Diseño

- **Los Patrones son diseño**
  - Pero: trascienden la idea de “identificar clases y asociaciones”

- **Durante el Diseño, los patrones ayudan a:**
  - Encontrar clases de diseño apropiadas
  - Determinar la granularidad de los objetos
  - Especificar interfaces
  - Especificar las implementaciones
  - Construcción de software flexible y reutilizable
  - Diseñar para el cambio

Patrones y Clases del Diseño

- *Los Patrones ayudan a encontrar clases de diseño*
- Los ***objetos del Dominio de aplicación*** representan conceptos del dominio relevantes al sistema
  - Son identificados durante la etapa de análisis por expertos del dominio, usuarios, etc.
  - Muchos objetos de diseño proceden del modelo de análisis que captura objetos del mundo real
- Durante el diseño OO aparecen ***clases del dominio de la solución***, que no tienen su equivalente en el mundo real
  - Son identificados por los desarrolladores

Patrones y Clases del Diseño

- *Los Patrones ayudan a encontrar clases de diseño*
- Las abstracciones que surgen durante el diseño son fundamentales para lograr un diseño flexible

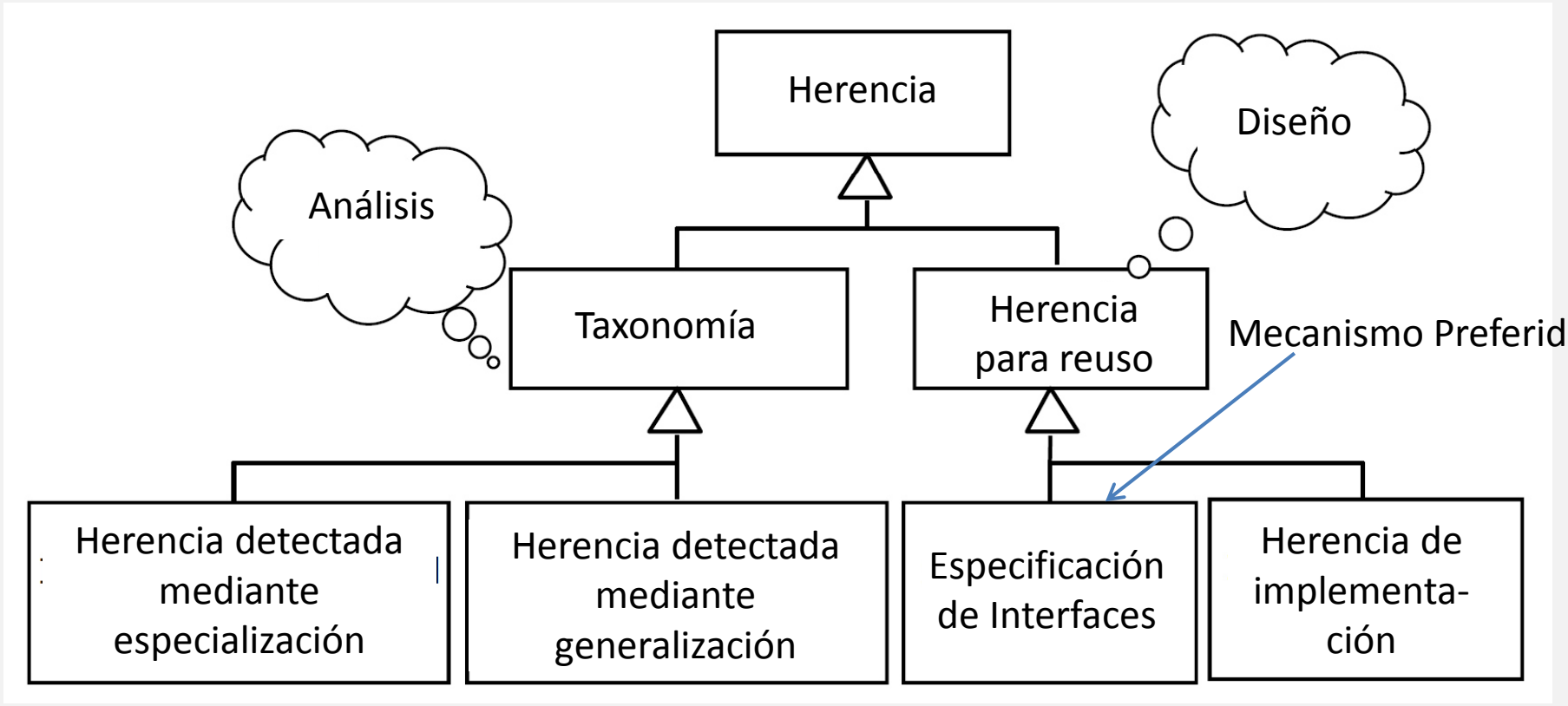
- Los Patrones ayudan a identificar abstracciones menos obvias y los objetos que las expresan (Ejemplo: Objetos que representan un algoritmo, un Estado, …)
- Estos objetos aparecen cuando se trata de obtener un diseño flexible y reusable >>>> *aplicando patrones*.

Patrones y Uso de la Herencia

- La herencia puede usarse con dos enfoques diferentes
  - Descripción de taxonomías
  - Especificación de Interfaces
- Identificación de taxonomías
  - Usada durante el análisis de requerimientos.
  - Actividad: identificar objetos del dominio de aplicación que están relacionados jerárquicamente
  - Meta: hace más comprensible el modelo de dominio.
- Especificación de interfaces
  - Usada durante el diseño de objetos
  - Actividad: especificar la ***signatura*** de los objetos identificados
  - Meta: incrementa *reusabilidad*, modificabilidad y extensibilidad

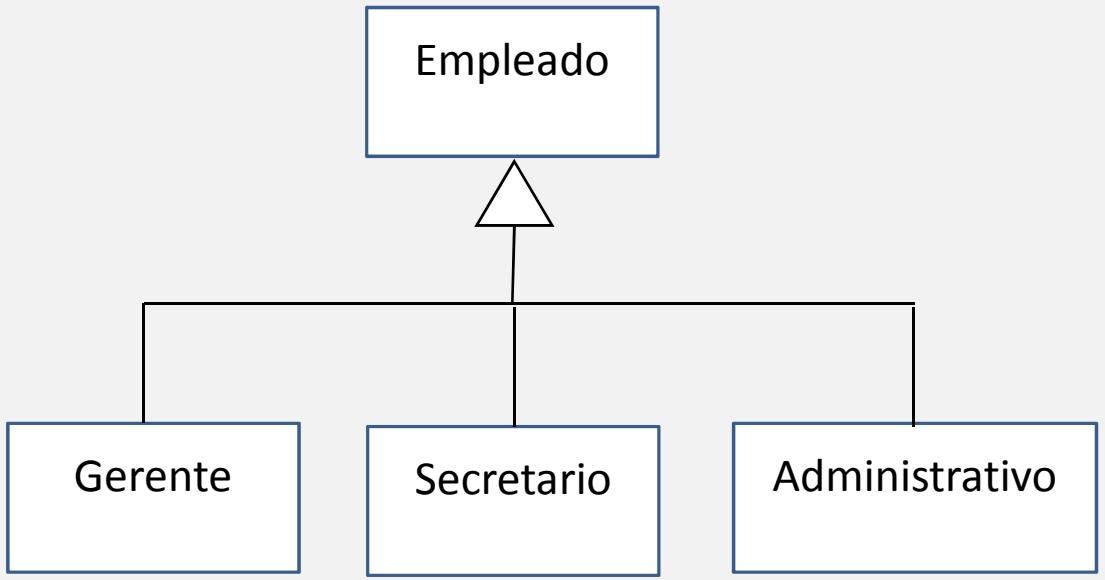
Patrones. Uso de la Herencia

- Uso de la herencia durante el análisis y el diseño de Objetos



Patrones. Uso de la Herencia

- Ejemplo de Taxonomía identificada durante el análisis



Patrones. Interfaces e Implementaciones

- **Diferencia entre clase de un objeto y su tipo**
- La **clase de un objeto** determina cómo se implementa un objeto
  - La clase define el estado interno y la implementación de las operaciones
- El **tipo de un objeto** sólo se refiere a su interfaz: conjunto de peticiones a las que puede responder
  - Un objeto puede tener muchos tipos y objetos de clases diferentes pueden tener el mismo tipo
- **Relación estrecha ente clase y tipo:** Dado que la clase define las operaciones del objeto, también define su tipo.

Interfaces e Implementaciones

- **Diferencia Herencia de Clases y Herencia de Interfaces:**
- La **herencia de clases** define la implementación de un objeto en términos de la implementación de otro objeto (Mecanismo para compartir código y representación)
  - Extiende la funcionalidad de las aplicaciones reusando funcionalidad en la clase padre (reuso de código)
- La **herencia de interfaces** (o subtipado) define cuándo se puede usar un objeto en lugar de otro
  - Se hereda de una clase abstracta o interfaz todas las operaciones especificadas, pero sin implementar
  - Se definen familias de objetos con interfaces idénticas. Una subclase añade o redefine operaciones pero no oculta operaciones de la clase padre
  - Todas las subclases pueden responder a las peticiones de la interfaz de su clase abstracta >>> subtipos de la clase abstracta

Interfaces e Implementaciones

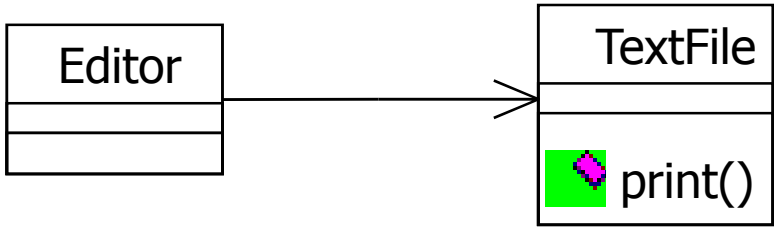
- ***Programar hacia la interfaz, no hacia la implementación***
- **Ventajas de manipular objetos en términos de la interfaz definida por las clases abstractas**:
  - Los clientes no tienen que conocer los tipos específicos de los objetos que usan, basta con que se conecten con la interfaz
  - Los clientes desconocen las implementaciones, sólo conocen las clases abstractas

- **Programar hacia la interfaz, no hacia la implementación**
  - No declarar variables de clases concretas sino abstractas.
  - Patrones de creación (se usarán para crear las clases concretas) permiten que un sistema esté basado en términos de interfaces y no en implementaciones.

Interfaces e Implementaciones

- ***Programar hacia la interfaz, no hacia la implementación***

Ejemplo: El Editor conoce a la clase concreta TextFile



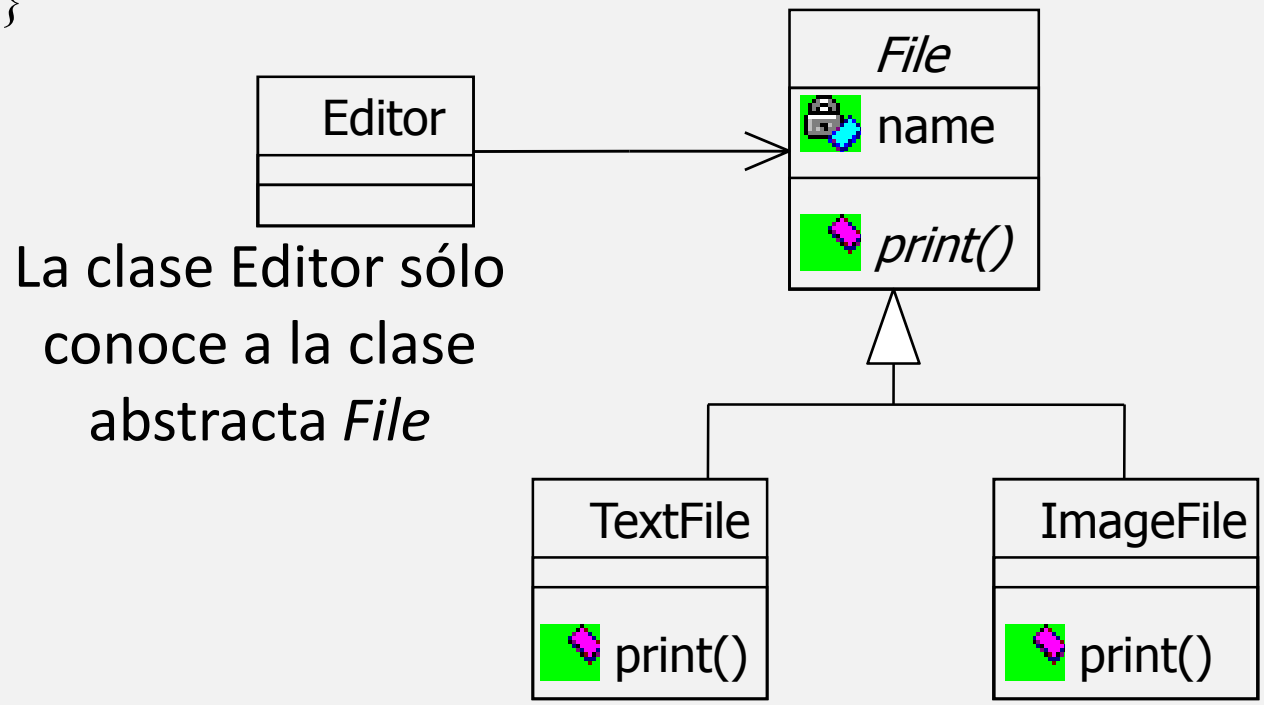
La clase Editor conoce
una implementación

Se diseña una solución mejor siguiendo la guía de diseño>>>

Interfaces e Implementaciones

- ***Programar hacia la interfaz, no hacia la implementación***

```
class Editor {
        public void handleFile (File f) {
                f.print(); //llamada polimórfica a print()
        }
}
```



La clase Editor sólo conoce a la clase abstracta *File*

Patrones y Reuso. Composición

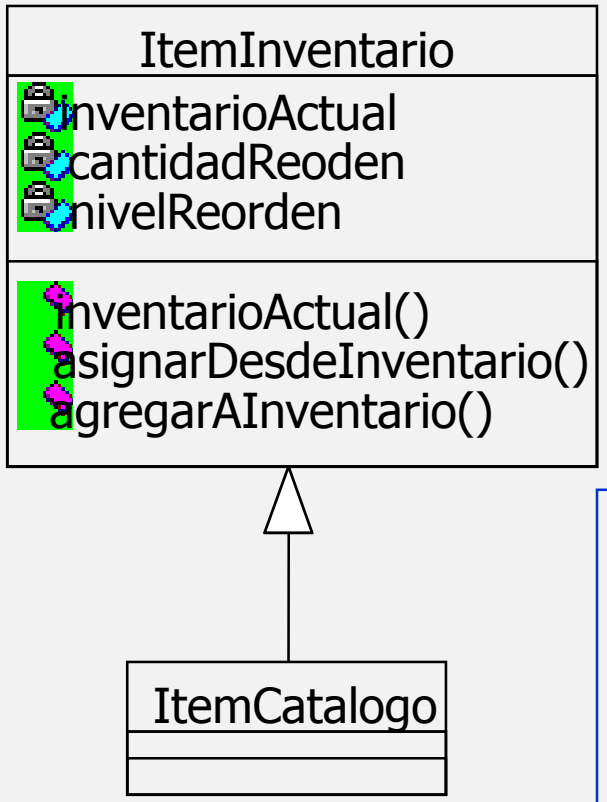
- ***Favorecer la composición de objetos frente a la herencia de clases***
- Dos técnicas comunes para reusar funcionalidad: herencia de clases y composición de objetos
  - **Herencia de clases**: Reuso caja blanca.
    - Es un mecanismo estático, se define en tiempo de compilación, de limitada flexibilidad.
  - **Composición de objetos**: Reuso caja negra: la nueva funcionalidad se obtiene ensamblando o componiendo objetos.
    - La composición se define dinámicamente en tiempo de ejecución a través de objetos que referencian otros objetos.

Patrones y Reuso. Composición

- ***Favorecer la composición de objetos frente a la herencia de clases***
- Ventajas de la composición:
  - Cualquier objeto puede ser reemplazado por otro en tiempo de ejecución mientras sean del mismo tipo.
  - Puesto que a los objetos se accede a través de sus interfaces, no se rompe su encapsulación.
  - Ayuda a mantener cada clase encapsulada y centrada en una sola tarea.
  - Se mantienen las jerarquías de clases pequeñas y más fáciles de manejar y mantener.
- *Los patrones favorecen el reuso a través de la composición en vez de la herencia (cuando es posible)*

Patrones y Reuso. Composición

- ***Favorecer la composición de objetos frente a la herencia de clases***



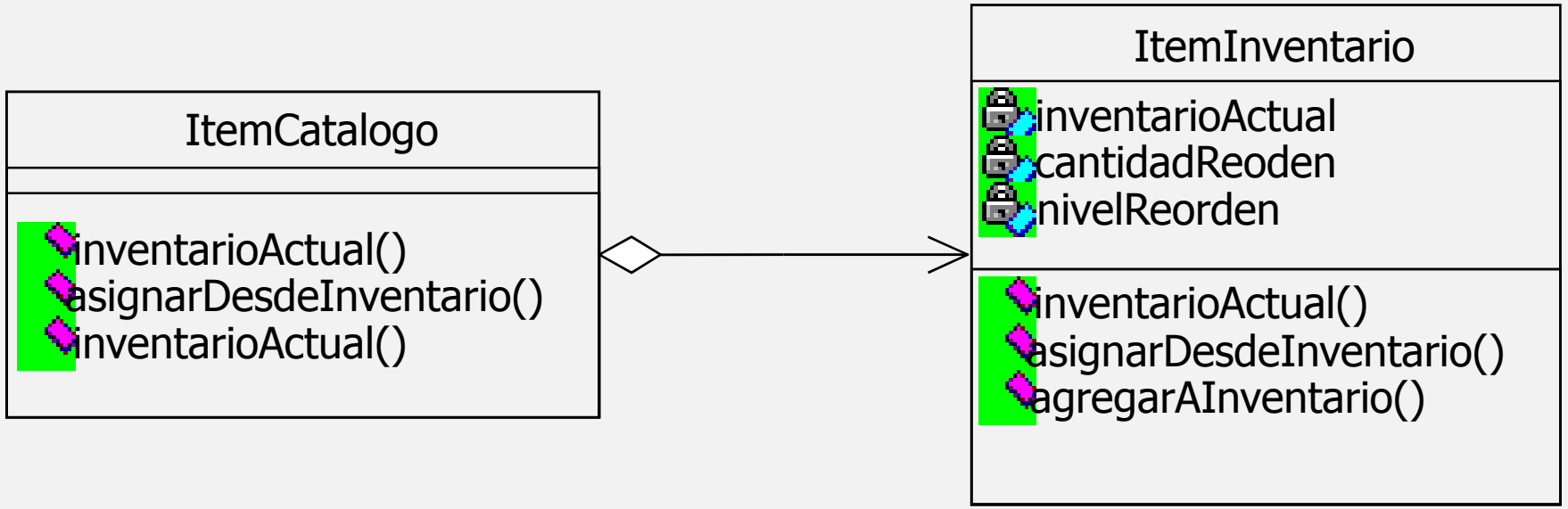
ItemCatalogo hereda sus propiedades de ItemInventario
Una vez creado de esta forma, durante la ejecución siempre será un item de este tipo.

Problema de usar Herencia: Si una Compañía determina que algunos items del Catálogo no deben retenerse en el Inventario sino ser obtenidos de proveedores sólo cuando sean ordenados, las instancias seguirán teniendo las propiedades heredadas de la clase Inventario

Patrones y Reuso. Composición

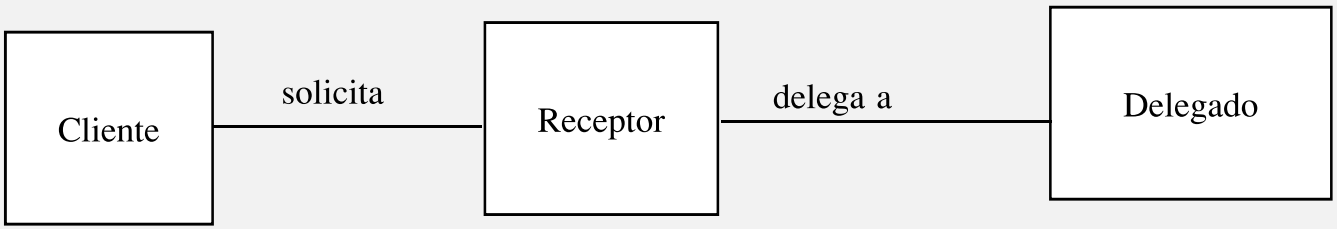
- ***Favorecer la composición de objetos frente a la herencia de clases***

Usando composición >>>> ItemCatalogo tiene una instancia ItemInventario como una parte opcional.
Algunos items tendrán esa parte y otros no.
Un objeto ItemCatalogo puede agregar y remover esa parte en tiempo de ejecución



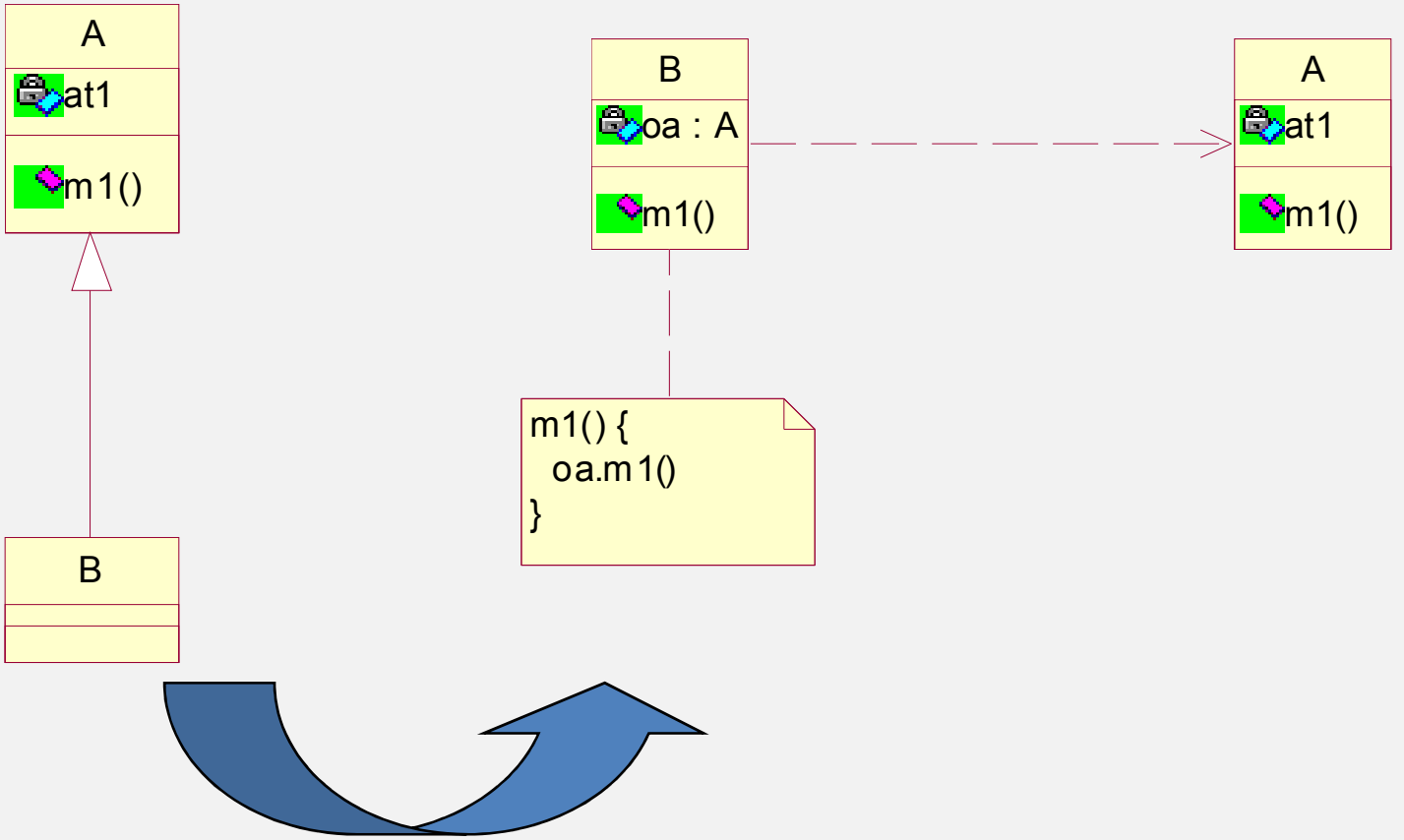
Patrones y Reuso. Delegación

- ***Los Patrones hacen un intenso uso de la Delgación***
- Es un modo de lograr gran potencia de la composición para el reuso.
- En la Delegación están implicados dos objetos en el manejo de un pedido:
  - Un objeto receptor delega operaciones a su delegado.



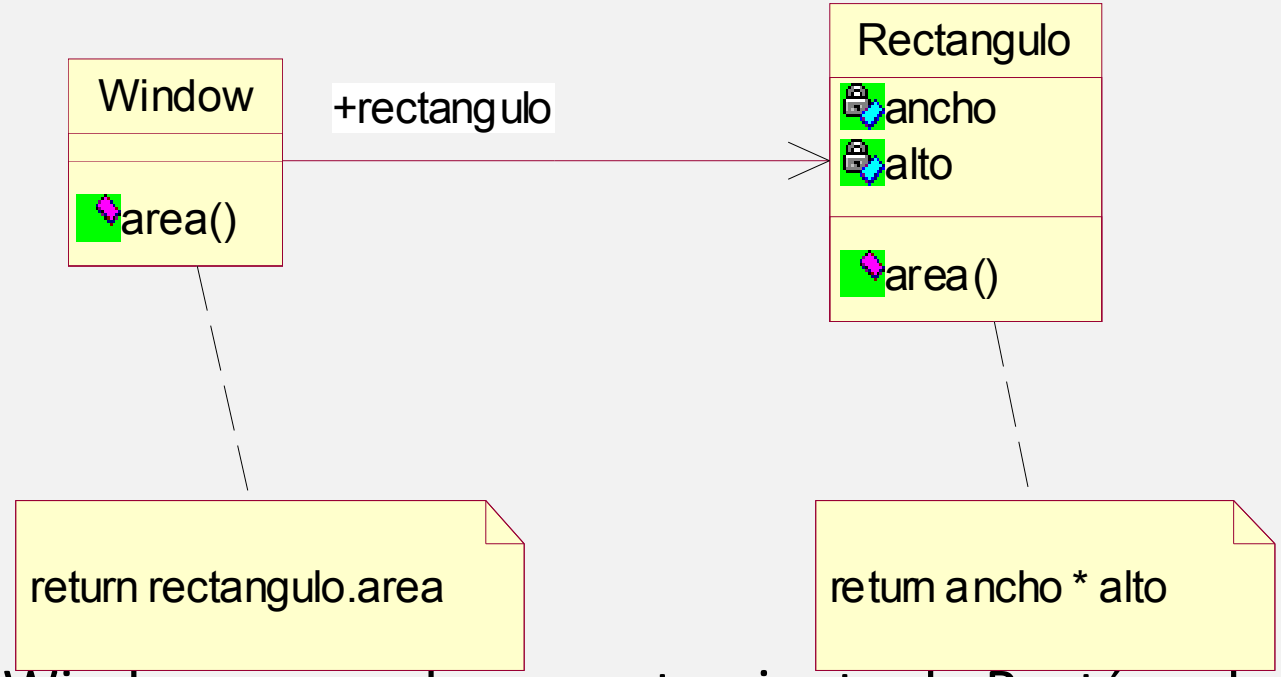
Delegación en lugar de Herencia

- En el primer caso B hereda m1() de A: mecanismo estático
- En el segundo caso, B delega en A el comportamiento m1() para usar cuando lo necesite: mecanismo dinámico



Delegación. Ejemplo

- Ejemplo Delegación: La clase Window delega en Rectángulo la responsabilidad de calcular el área (en lugar de que Ventana herede todo el comportamiento de Rectángulo).



- La clase Window reusa el comportamiento de Rectángulo
  - guarda una instancia de ésta
  - delega en ella el comportamiento específico de un rectángulo

Diseño para el cambio. Frameworks

- ***Frameworks: Proveen reuso de diseño de alto nivel***
- Un *framework* es una colección organizada de clases cooperantes que constituyen un diseño reutilizable *para una clase específica de software*.
- Establece la **arquitectura de la aplicación**: Provee un conjunto de subsistemas predefinidos, especifica sus responsabilidades e incluye reglas y guías para organizar las relaciones entre los mismos (Buschmann *et al*, 1996).
- Instanciación: Adaptación del framework a una aplicación particular.
- La reutilización se implementa entonces mediante la instanciación del framework en una aplicación concreta.
- Los beneficios clave de los frameworks son: reusabilidad y extensibilidad

Diseño para el cambio. Frameworks

Características

- Un framework ofrece un conjunto integrado de funcionalidad específica de un dominio
  - P.ej.: aplicaciones financieras, servicios de telecomunicación, sistemas de ventanas, aplicaciones distribuidas, aplicaciones web
- Los frameworks invierten el control en ejecución entre la aplicación y el software sobre el que está basada
  - El framework determina qué métodos se invocan en respuesta a eventos (se reusa el código del cuerpo principal y se escribe el código al que llama)
- Un framework es una aplicación a medio-terminar
  - Las aplicaciones completas se desarrollan mediante herencia, e instanciando componentes parametrizados del framework
- Los patrones de diseño ayudan a que la arquitectura del framework sea reutilizada en muchas aplicaciones diferentes sin necesidad de rediseño

Clasificación de Frameworks

- **Generales o Middleware**: Encapsulan una capa de funcionalidad horizontal que se puede aplicar en la construcción de una gran variedad de programas.
  - Programación de GUIs
  - Entornos de programación visual
- **De soporte o de base**: Proporcionan servicios básicos a nivel de sistema.
  - Infraestructuras de comunicaciones
  - Plataformas de componentes
- **De dominio o de Empresa**: Aplicables a un dominio de aplicación específico o línea de producto. Deben adaptarse a las áreas de negocio para las que están diseñados a medida (Telecomunicaciones, Ingeniería financiera, etc.).

Patrones vs. Frameworks

- Los patrones de diseño tienen descripciones más abstractas que los frameworks
  - Las descripciones de patrones suelen ser independientes de los detalles de implementación o del lenguaje de programación (salvo ejemplos usados en su descripción)
  - Los frameworks están implementados en un lenguaje de programación, y pueden ser ejecutados y reutilizados directamente
- Los patrones de diseño son elementos arquitecturales más pequeños que los frameworks
  - Un framework incorpora varios patrones
  - Los patrones se pueden usar para documentar frameworks
- Los patrones de diseño están menos especializados que los frameworks
  - Los frameworks siempre se aplican a un dominio de aplicación particular

Bibliografía

- Agile software Development. Principles, Patterns and Practices. Martin R. Prentice Hall , 2003
- Design Patterns: Elements of Reusable Object-Oriented Software. Gamma E, Helm R. Johnson R. y Vlissides J. Addison Wesley, 1995-2005
- Agile Software Development and Reusability. Singh J., Singh A. Ed. IJREAS. Vol 2. Issue 2, Feb 2012