UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Departamento de Informática
Universidad Técnica Federico Santa María

# Reuso + Desarrollo Basado en Componentes (CBD)

Ingeniería de Software

**Hernán Astudillo & Gastón Márquez**
*Departamento de Informática*
*Universidad Técnica Federico Santa María*

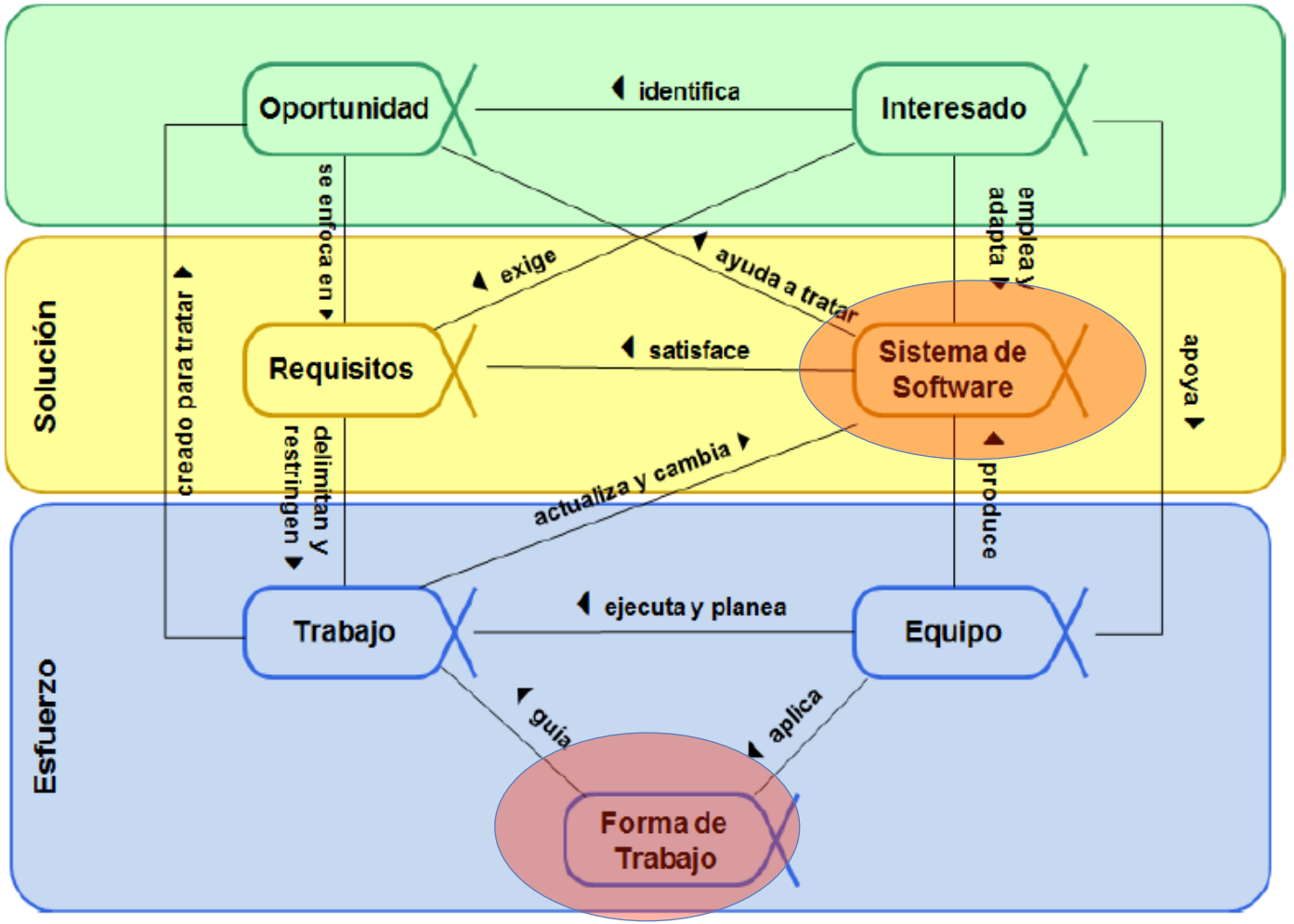
# Diseño y reuso

- Técnicas de diseño han sido hechas para hacer sistemas desde cero
  - …pero la mayoría de los sistemas reales son modificaciones o reemplazos de otros sistemas
- Uso de Componentes COTS
  - “Commercial Off-the-shelf Components”
  - Comerciales o libres (open source)
- El problema del diseñador es diferente al problema del programador
  - un problema de optimización vs
  - un problema de selección

# Escalas de reuso

- Componentes – Piezas reutilizables
  - Software empaquetado
- "Framework" para un dominio
  - Software + diseño
- Patrones de diseño
  - Heurísticas sistematizadas
- Arquitecturas de referencia y modelos de Componentes
  - Vocabulario y permiten intercambio ínter-organizacional

# Contexto


Oportunidad
identifica
Interesado
se enfoca en
exige
ayuda a tratar
emplea y adapta
Solución
creado para tratar
Requisitos
satisface
Sistema de Software
apoya
delimitan y restringen
actualiza y cambia
produce
Trabajo
ejecuta y planea
Equipo
Esfuerzo
guía
aplica
Forma de Trabajo

# Piezas reutilizables

# Componentes

- Componente [Whitehead]
  - Pieza separable (independiente del contexto) de software ejecutable
  - ...que tiene sentido como unidad
  - ...y puede interoperar con otros componentes
  - ...dentro de un ambiente de apoyo
  - ...y es accesable sólo vía sus interfaces
  - ...y está listo para usar (aparte de instalación y configuración)

# Productos

- Categorías de productos
  - Empaquetados por tipo de problema y de solución
- Tipos de solución: tecnologías
  - “Middleware”
  - MOM, BD, “directory servers”, monitores transaccionales, “workflow”...
- Tipos de problemas: servicios del negocio
  - Paquetes
  - ERP (Enterprise Resource Planning), CRM (Customer Relationship management)...

# Diseños Reutilizables

- Diseños reutilizables
  - “Frameworks”
    - aplicaciones o sistemas incompletos
  - Estilos
    - formas típicas de sistemas
  - Patrones
    - heurísticas para solución con formas típicas
  - Líneas de productos
    - generalizaciones de aplicaciones exitosas

# Object-Oriented Frameworks [1]

- "Framework" orientado a objetos
  - diseño reusable…
  - que modela parte de un sistema de software…
  - con un conjunto de clases abstractas...
  - que encarnan la colaboración entre sus instancias
- Forma de (re)uso
  - "hot spots": clases abstractas
  - construir sub-clases específicas al sistema

# Object-Oriented Frameworks [2]

- Resultan de procesos de análisis de dominio
- Reducir el esfuerzo de desarrollo porque permiten reusar diseño y código
- Difícil desarrollarlos porque deben ser fáciles de usar y tener poder expresivo para cubrir variaciones del dominio

# Ejemplos de frameworks [1]

- ANT
  - Herramienta open source utilizada en la compilación y creación de programas Java
  - Escrito en XML y Java \ ofrece una solución interoperable al nivel de sistema operativo (gracias a Java) y configuraciones descriptivas (gracias a XML)
  - Se usa creando nuevas clases que especializan las clases del framework
  - http://ant.apache.org/
- Ampliamente usado por desarrolladores Java (¡aunque no sepan qué es un framework!)

# Ejemplos de frameworks [2]

- Hibernate
  - Persistencia objeto/relacional y servicio de consultas para Java y .NET
  - Permite desarrollar clases siguiendo el paradigma OO y expresar consultas en HQL (extensión propia y portable de SQL), en SQL nativo, o con un criterio OO
  - http://www.hibernate.org/
- Ampliamente usado por proyectos open source y propietarios

# Ejemplos de frameworks [3]

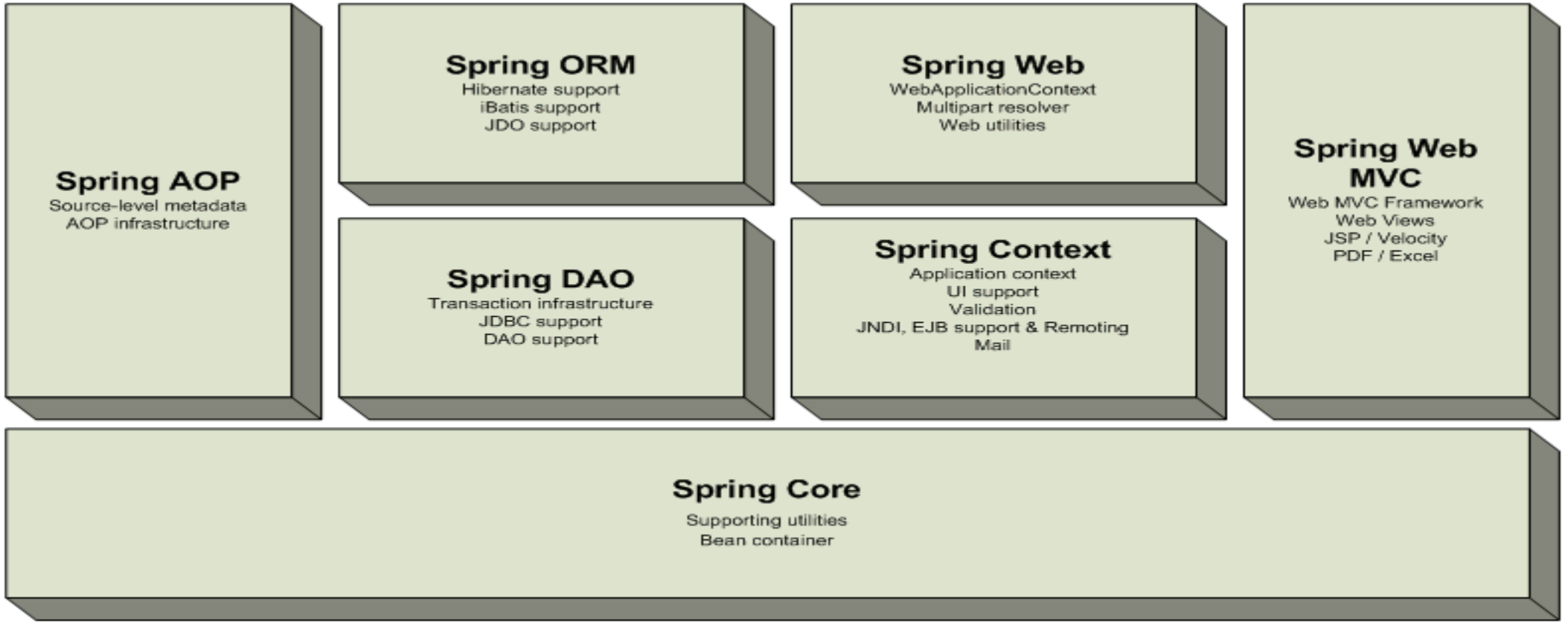
- Struts
  - Framework y toolkit Open Source que ayuda a desarrolladores a construir aplicaciones para la Web
  - Arquitecturas de aplicación basadas en el enfoque “Model 2”, una variación del paradigma de diseño clásico MVC
    - Usando Model 2, un servlet (o equivalente) maneja la ejecución de la lógica de negocio, y la lógica de presentación reside principalmente en “server pages”
  - http://struts.apache.org/

# Ejemplos de frameworks [4]

- Spring
  - Sistema para ensamblar componentes vía archivos de configuración
  - Puede ser usado en conjunto con otros frameworks
  - Es modular y ha sido dividido lógicamente en paquetes independientes, los que pueden funcionar independientemente
  - Está basado en el patrón Inversión of Control/Dependency Injection
  - Para J2EE: http://www.springframework.org/
  - Para .NET: http://www.springframework.net/

# Ejemplos de frameworks [5]

- Spring



# Catálogos de componentes

- En la actualidad, se considera mala práctica desarrollar una aplicación entera desde cero (salvo que sea muy especial)
  - Lo primero es buscar componentes o frameworks que sea posible reusar
- Hay tantos componentes, que se requiere organizarlos
  - Existen numerosos catálogos de componentes (ver próximas transparencias)

# Catálogos [1]

- SourceForge
  - https://sourceforge.net
  - Un sitio Web de desarrollo de software open source, ofrece hosting a más de 100.000 proyectos y tiene sobre 1.000.000 de usuarios registrados con manejo de proyectos, versiones, comunicación y código centralizado
  - Gran repositorio de código y aplicaciones open source de distintas categorías (clustering, BD, juegos, empresariales, multimedia, financieros, seguridad, SysAdmin, VoIP,…)

# Catálogos [2]

- FreeCode (era FreshMeat)
  - http://www.freecode.com
  - Gran índice de software open source para Unix y cross-platform, temas visuales, y software para PalmOS

# Catálogos [3]

- ObjectWeb
  - http://asm.ow2.org
  - Comunidad de software open source cuyo objetivo es el desarrollo de middleware distribuido open source, en la forma de componentes adaptables y flexibles
  - Componentes van desde frameworks de software específicos a protocolos para plataformas integradas
  - Creada a fines de 1999 por Bull, France Telecom R&D e INRIA (hosting provisto por INRIA); en 2003 pasó a ser un consorcio internacional

# Patrones de diseño

# Patrones de Diseño

- Definición
  - “An object-oriented design pattern describes communicating objects and classes that are customized to solve a generic design problem in a particular context” [Gamma95]
- Ventajas
  - Idea abstracta reusable
  - Vocabulario de comunicación
  - Bloques básicos
  - Capturan “mejores prácticas” de diseño
  - Heurísticas sistematizadas

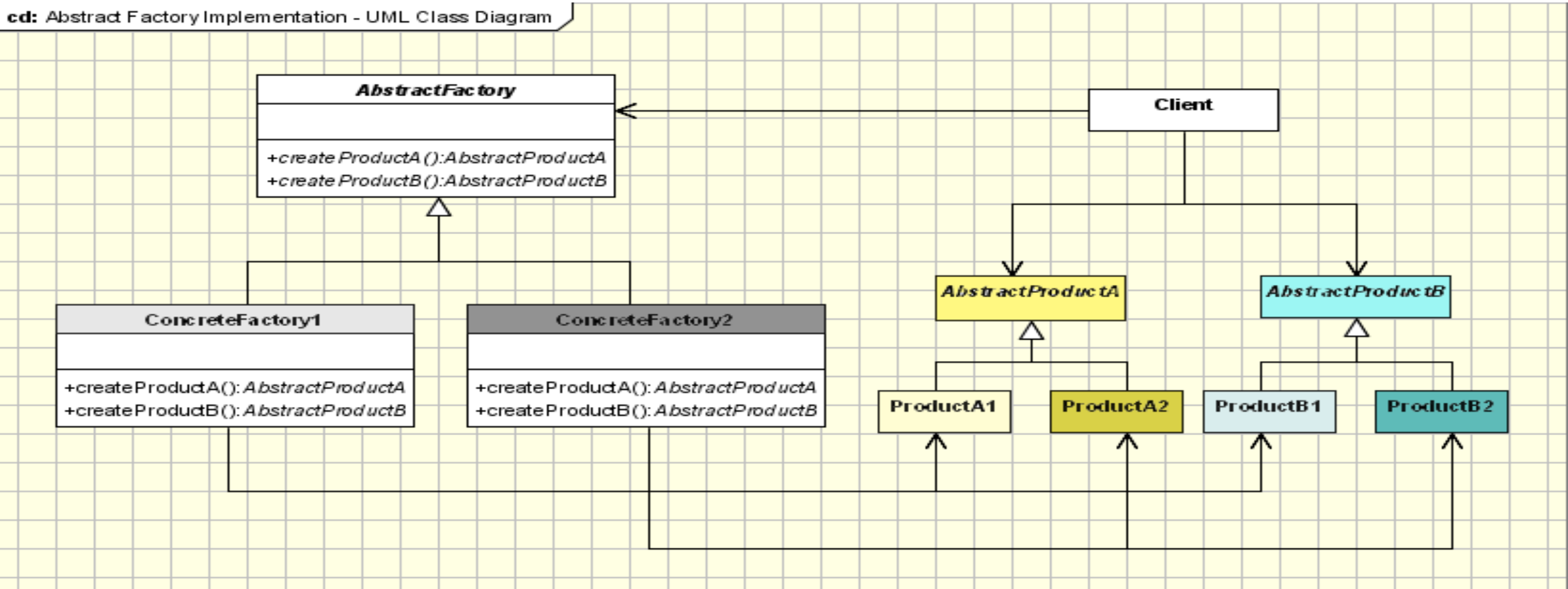
# Abstract Factory [1]

- Patrón de diseño creacional
- Propósito: proveer una interfaz para crear familias de objetos dependientes sin especificar (al cliente) sus clases concretas
- Una clase abstracta provee los métodos (a modo de interfaz) que son comunes para todos los tipos de productos
- Varias clases (que representan los tipos de productos) heredan los métodos desde la clase abstracta

# Abstract Factory [2]

- La instanciación de la clase puede ser entendida como “crear un producto en una fábrica”
  - Esto es, entenderemos al objeto como un producto
- Idea central: el cliente, en vez de crear directamente el objeto utilizando el operador adecuado (ej: en JAVA el operador new), entrega información a la fábrica sobre el tipo de producto que necesita
  - La fábrica instancia una nueva clase (crea el objeto/producto) y lo devuelve al cliente
  - El cliente utiliza al objeto como un “producto abstracto” sin interesarse en la implementación concreta

# Abstract Factory [3]



# Abstract Factory [4]

- Las clases que participan en el patrón son las siguientes:
  - AbstractFactory: declara una interface para los operadores que crean productos abstractos
  - ConcreteFactory: implementa las operaciones para crear productos concretos
  - AbstractProduct: declara la interface para un tipo de producto objeto
  - Product: define un producto a ser creado por el correspondiente ConcreteFactory, es implementado por la interface AbstractProduct
  - Client: usa la interface declarada por la clases AbstractFactory y AbstractProduct

# Abstract Factory [5]

- El hecho de que la fábrica devuelve una referencia abstracta al objeto creado significa que el cliente no tiene conocimiento del tipo de objeto.
- Una consecuencia de lo anterior es que cuando se necesitan nuevos tipos concretos de objetos, todo lo que tenemos que hacer es modificar el código de cliente y hacer uso de una fábrica diferente, que es mucho más fácil que crear instancias de un nuevo tipo.

# Abstract Factory [6]

- La clase AbstractFactory es la que determina el tipo real del objeto concreto y lo crea, pero devuelve una referencia abstracta al objeto concreto recién creado.
- Esto determina el comportamiento del cliente que pide a la fábrica la creación de un objeto de un cierto tipo abstracto y para devolver la referencia a él, manteniendo el cliente de saber nada acerca de la creación real del objeto.

# Abstract Factory [7]

```
abstract class AbstractProductA{
        public abstract void operationA1();
        public abstract void operationA2();
};
class ProductA1 extends AbstractProductA{
        ProductA1(String arg){
                System.out.println("Hello "+arg);
        } // Implement the code here
        public void operationA1() { };
        public void operationA2() { };
};
class ProductA2 extends AbstractProductA{
        ProductA2(String arg){
                System.out.println("Hello "+arg);
        } // Implement the code here
        public void operationA1() { };
        public void operationA2() { };
};
abstract class AbstractProductB{
        //public abstract void operationB1();
        //public abstract void operationB2();
};
```

# Abstract Factory [7]

```
class ProductB1 extends AbstractProductB{
        ProductB1(String arg){
                System.out.println("Hello "+arg);
        } // Implement the code here
};
class ProductB2 extends AbstractProductB{
        ProductB2(String arg){
                System.out.println("Hello "+arg);
        } // Implement the code here
};
abstract class AbstractFactory{
        abstract AbstractProductA createProductA();
        abstract AbstractProductB createProductB();
};
class ConcreteFactory1 extends AbstractFactory{
        AbstractProductA createProductA(){
        return new ProductA1("ProductA1");
        }
        AbstractProductB createProductB(){
        return new ProductB1("ProductB1");
        }
};
```

# Abstract Factory [7]

```
class ConcreteFactory2 extends AbstractFactory{
        AbstractProductA createProductA(){
                return new ProductA2("ProductA2");
        }
        AbstractProductB createProductB(){
                return new ProductB2("ProductB2");
        }
};
//Factory creator - indirect way of instantiating the factories
class FactoryMaker{
        private static AbstractFactory pf=null;
        static AbstractFactory getFactory(String choice){
                if(choice.equals("a")){
                        pf=new ConcreteFactory1();
                }else if(choice.equals("b")){
                                        pf=new
ConcreteFactory2();
                        } return pf;
        }
}
```

# Abstract Factory [7]

```
// Client
public class Client{
        public static void main(String args[]){
                AbstractFactory
pf=FactoryMaker.getFactory("a");
                AbstractProductA product=pf.createProductA();
        //more function calls on product
        }
}
```

# ¿Cuándo ocupar Abstract Factory?

- Cuando el proyecto en el cual estamos trabajando el sistema debe ser independiente a la forma en que los productos trabajan cuando son creados
- Cuando el sistema debe ser configurado para trabajar con varias familias de productos
- Cuando una familia de productos está diseñada para funcionar todos juntas
- Cuando se necesita la creación de una librería de productos, donde es solo relevante la interface, pero no la implementación

# Patrones de aplicaciones corporativas

# Patrones de Aplicaciones Corporativas

- "Patterns of Enterprise Application Architecture"
  - Fowler et al. [Fowler 2002]
- Presumen una arquitectura convencional de 3 capas
  - dominio (negocio)
  - datos
  - presentación (Web)
- Ofrecen soluciones para 3 tipos de tecnología
  - concurrencia
  - sesiones y estados
  - distribución

# Asumen arquitectura 3 capas

# Patrones de concurrencia

- Control de concurrencia
  - optimista vs. pesimista
- Transacciones

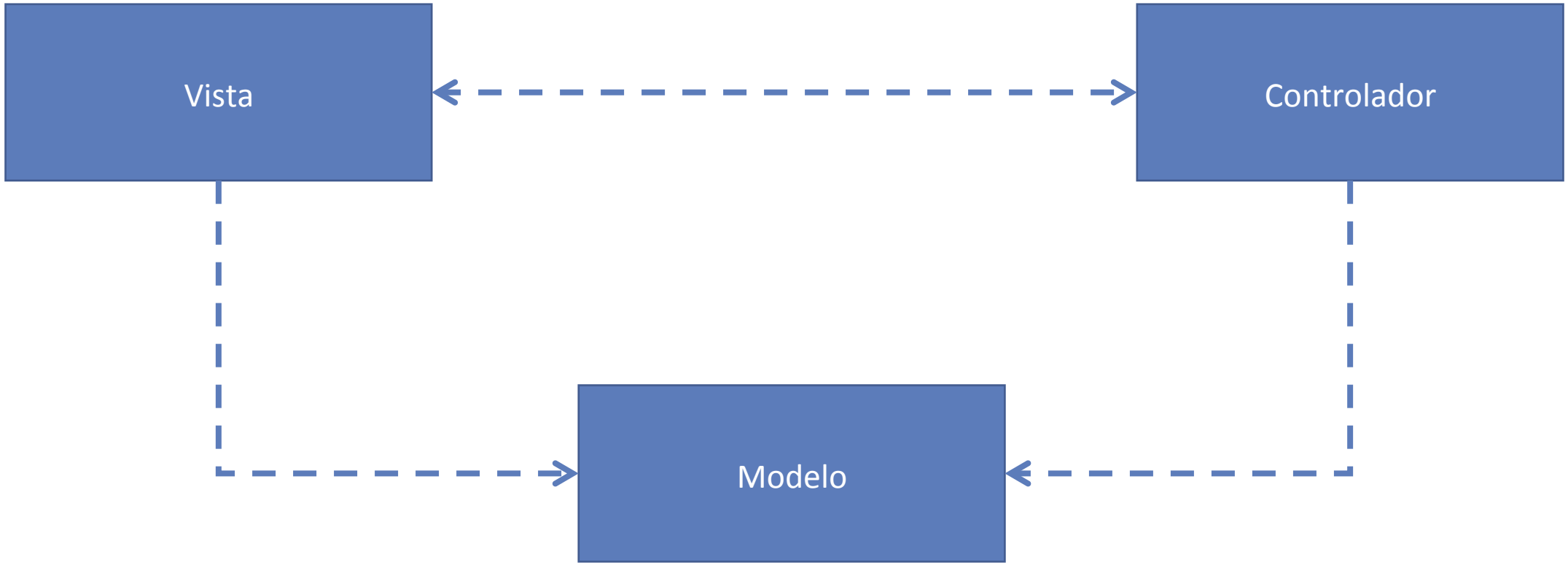
# Patrones de datos

- Mapeos relacional-objetos
  - campo identidad
  - clave extranjera
  - tabla de asociaciones
  - mapeo de dependencias
  - valor embutido
  - herencia en una tabla
  - clase en una tabla
- Metadatos

# Patrones de presentación

- MVC
- Vistas:
  - con plantillas
  - transformaciones
  - 2 pasos

# Dependencias entre componentes de MVC



# Separación Vista - Modelo

- Separación esencial en el patrón
  - Dependencia unidireccional: la vista depende del modelo
- Ventajas se derivan de:
  - Satisface distintos intereses técnicos
    - Presentación se preocupa de mecanismos de UI
    - Modelo se preocupa de políticas y reglas del negocio
  - Satisface distintos intereses de usuarios
    - Mismo modelo, pero con distintos públicos objetivos
      - Ej: Vista operacional versus vista estratégica

# Separación Vista - Controlador

- Separación “menos importante”
- En la práctica, la mayoría de las aplicaciones tienen un controlador por vista…
  - … pero esto no es una obligación
- Ventaja se deriva de:
  - Podemos tener dos controladores para una vista
    - Un controlador se encarga de tareas de edición
    - Otro controlador se encarga de tareas de sólo lectura

# MVC… ¿patrón de diseño?

- MVC es un patrón de diseño que por sí solo es demasiado grande
- Cumple con la definición que hemos dado de un patrón de diseño…
  - Describe un problema recurrente
  - Entrega una propuesta de solución genérica aplicable a este problema recurrente
- Sin embargo, necesitamos más ayuda (de otros patrones) para construir un sistema MVC

# Patrones de sesión y estado

- Estado de sesión en el cliente
- Estado de sesión en el servidor
- Estado de sesión en base de datos

# Resumen

- Problema del diseñador y programador difieren
  - un problema de optimización vs
  - un problema de selección
- Soluciones: piezas reutilizables
  - COTS
    - Modelos de componentes
  - Frameworks
- Soluciones mejores: diseños reutilizables
  - Patrones de diseño (aplicaciones incompletas)
  - Patrones de aplicaciones corporativas

# Patrones: Resumen

- Avance notable
  - ...desde reuso de código
  - ...a reuso de código con diseño
  - ...a reuso de diseños
- Patrones
  - Frutos de la experiencia
  - Heurísticas sistematizadas

# Patrones: Resumen

- Avance notable
  - ...desde “estilos” descriptivos
  - ...a “patrones” clásicos de sistemas
  - ...a patrones “dimensionales”
- Patrones
  - Frutos de la experiencia
  - Heurísticas sistematizadas

UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Departamento de Informática
Universidad Técnica Federico Santa María

# Reuso + Desarrollo Basado en Componentes (CBD)

Ingeniería de Software

**Hernán Astudillo & Gastón Márquez**
*Departamento de Informática*
*Universidad Técnica Federico Santa María*