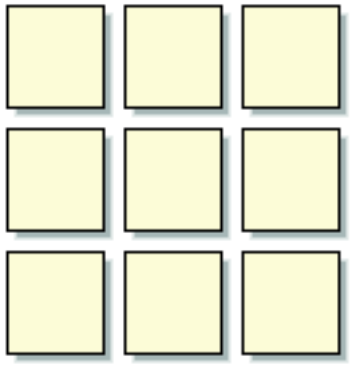
Patrones de Diseño: Model-View-Controller (MVC)

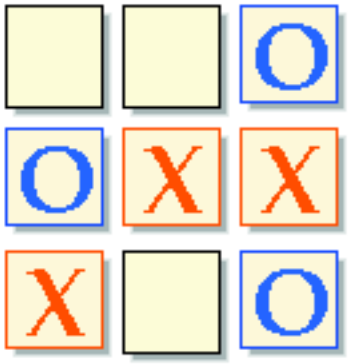
Análisis & Diseño de Software/Fundamentos de Ingeniería de Software

Cristian Orellana-Pablo Cruz Navea – Gastón Márquez-Hernán Astudillo
Departamento de Informática
Universidad Técnica Federico Santa María

Veamos un juego conocido

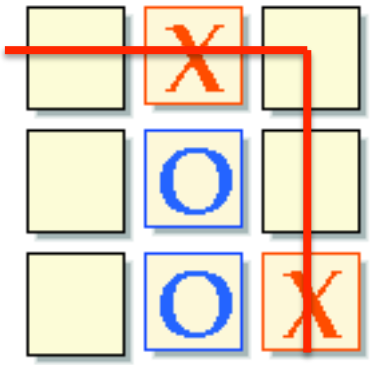


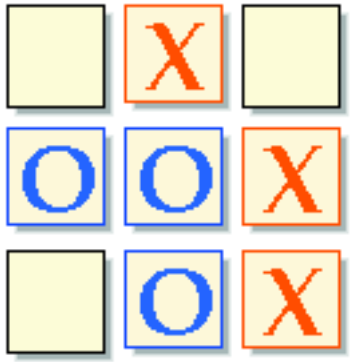
YOU ARE O



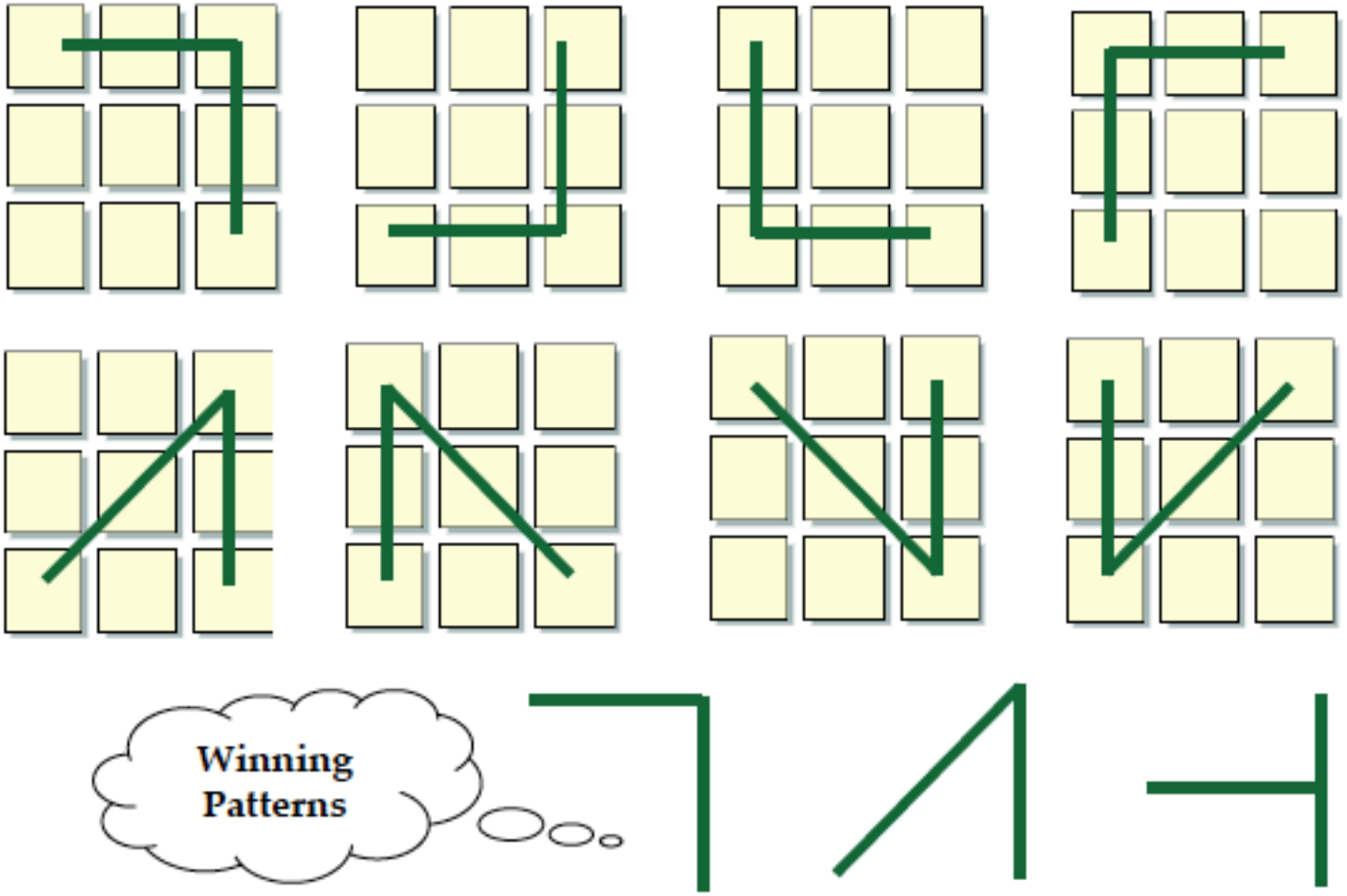
YOU ARE O

Veamos un juego conocido





Veamos un juego conocido



Veamos un juego conocido

- Parece ser que jugar al juego “Gato” suele ser fácil
- Pero, ¿Por qué?

Veamos un juego conocido

- Si usted conoce los posibles patrones que puede tener este juego, se puede *anticipar* a la jugada del oponente
- Por lo tanto, usted eventualmente podría abordar todos los posibles escenarios del juego y ganar

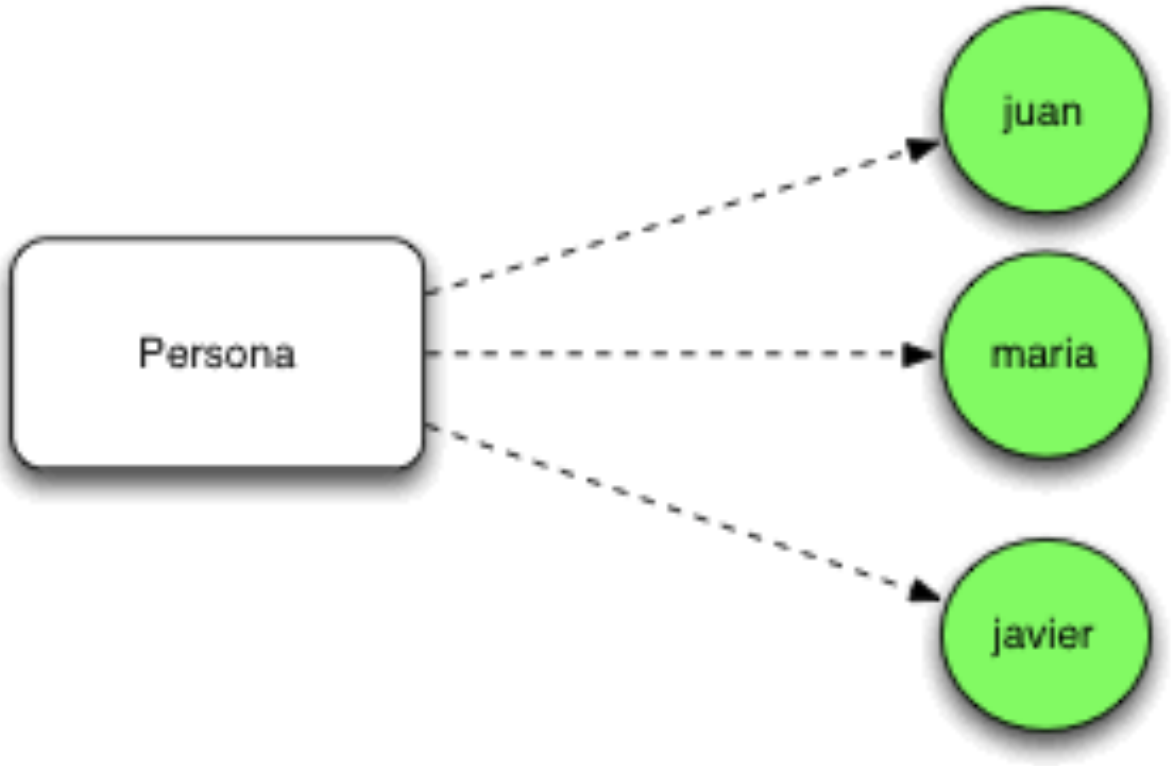
Patrones de Diseño [1]

- Los Patrones de Diseño son soluciones a problemas de diseño de software que se encuentran en la vida real de forma **recurrente** en el desarrollo de aplicaciones. Los Patrones tratan con código reusable e interacción con objetos.
- Además, para que una solución propuesta sea un Patrón, debe contener ciertas características.
  - Reusable
  - Efectivo

Patrones de Diseño [2]

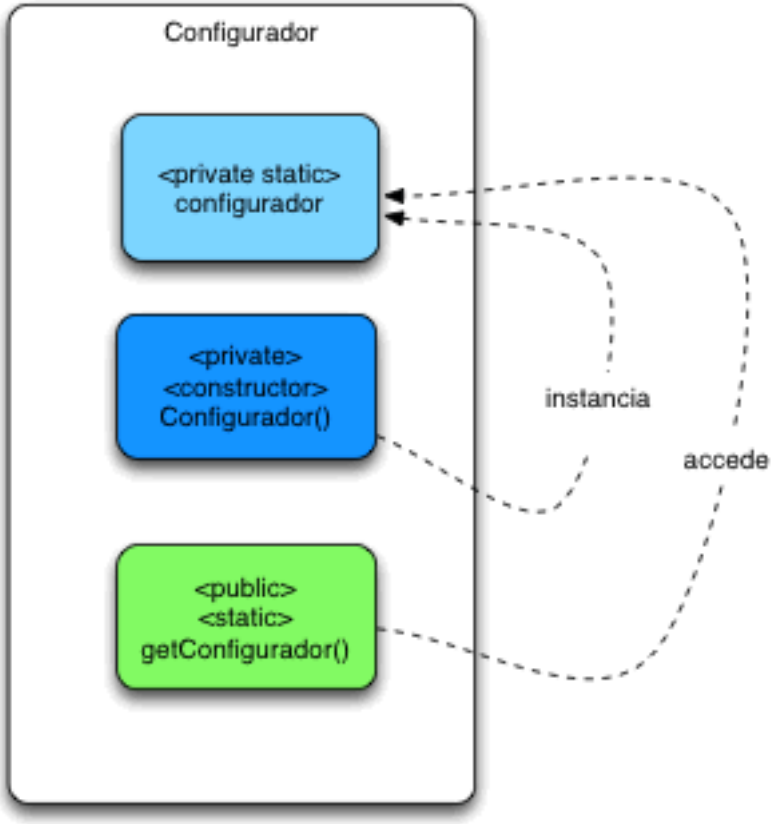
- Creational Patterns
  - Singleton
  - Factory
  - Abstract Factory
  - …
- Behavioral Patterns
  - Interpreter
  - Strategy
  - Memento
  - Observer
  - …
- Structural Patterns
  - Adapter
  - Bridge
  - …

Ejemplo Singleton [1]

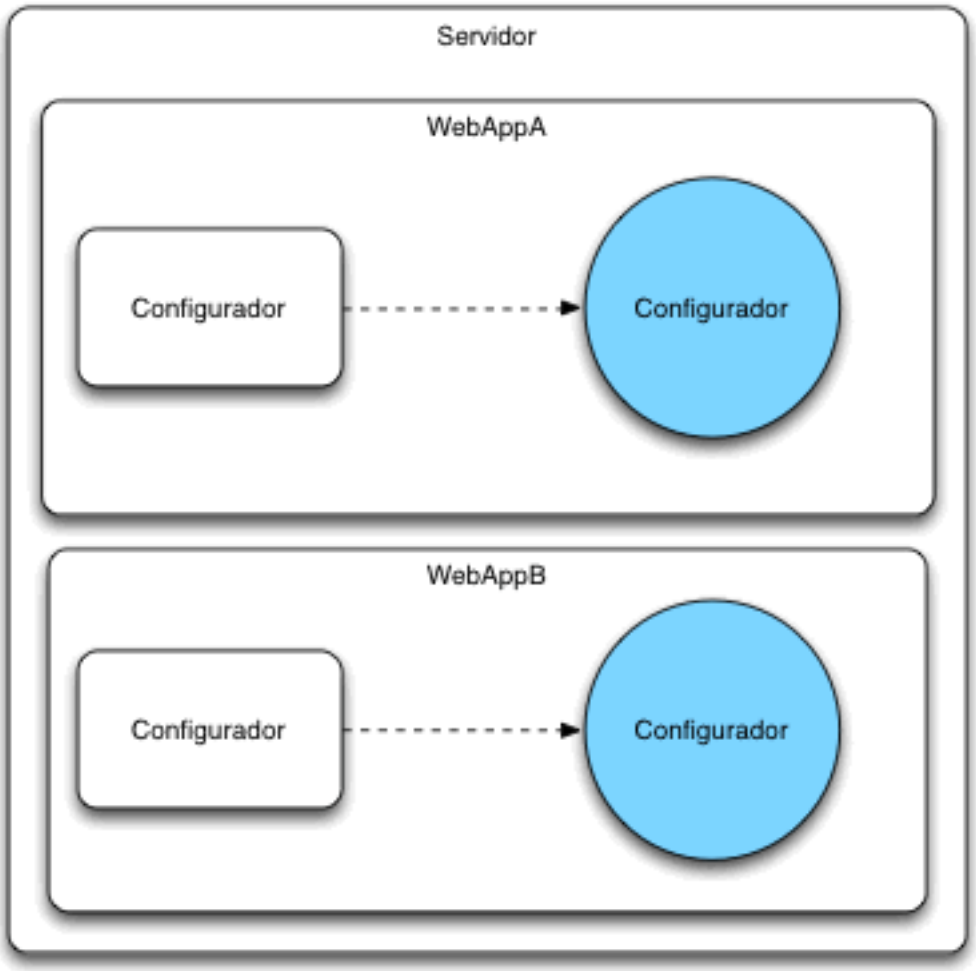


Ejemplo Singleton [2]

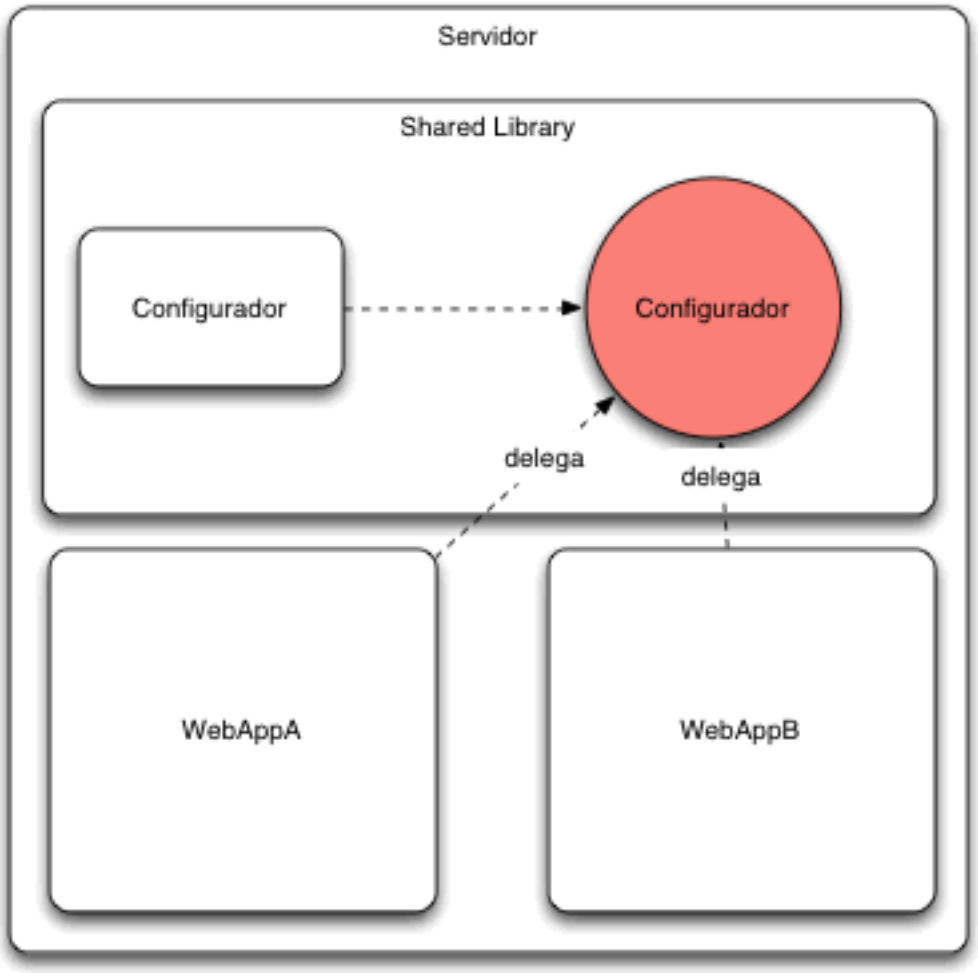
Ejemplo Singleton [3]



Ejemplo Singleton [4]



Ejemplo Singleton [5]



Model-View-Controller (MVC) [1]

- Patrón de diseño muy utilizado y probado en el mundo de las aplicaciones gráficas
- No es propiedad de las aplicaciones web
  - Pero son estas aplicaciones las más comunes cuando hablamos del patrón MVC
- Problema: la utilización de una sola clase para aplicaciones que manejan datos y elementos gráficos termina en trabajo confuso y difícil de trabajar por equipos de desarrollo
- Solución: separar las partes constituyentes de este tipo de aplicaciones en tres elementos:
  - Modelo
  - Vista
  - Controlador

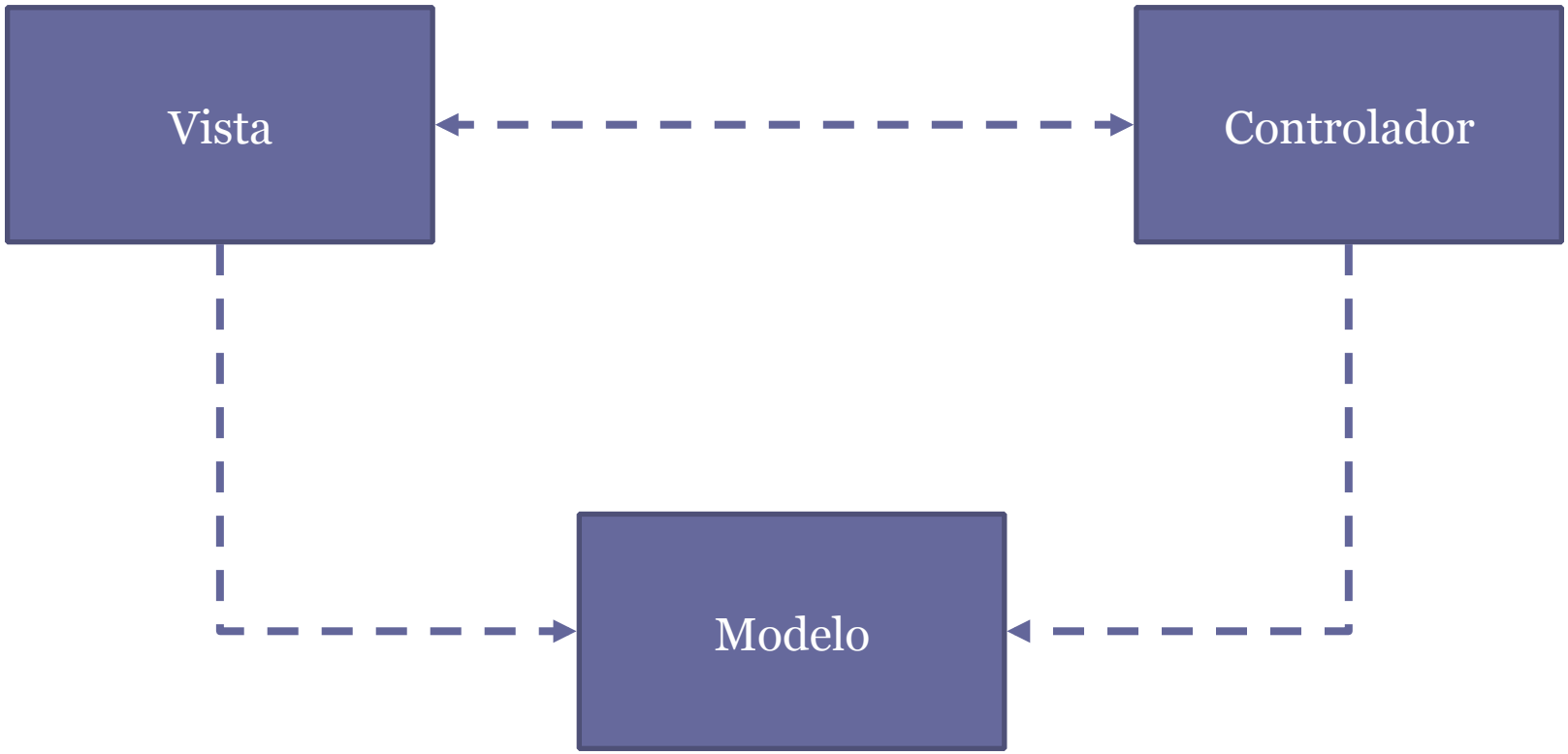
Model-View-Controller (MVC) [2]

- Modelo
  - Encapsula el “estado” de la aplicación
  - Responde consultas sobre el “estado” de la aplicación
  - Expone funcionalidad de la aplicación
  - Notifica a la Vista de cambios
- Vista
  - “Solicita actualizaciones desde el modelo
  - Envía “entradas del usuario” a los controladores
  - Permite al controlador la selección de una vista
- Controlador
  - Define el comportamiento de la aplicación
  - “Mapea” entradas del usuario a actualizaciones en el modelo
  - Selecciona una vista para responder
  - “Uno por cada funcionalidad”

Un poco de historia…

- Aparece por primera vez en 1970 como un *framework* desarrollado por Trygve Reenskaug
- Desarrollado para la plataforma Smalltalk
- Desde ese momento, se convirtió en un patrón de influencia mayor en la mayoría de los *frameworks* que involucran interfaces para usuarios
- Actualmente, muy citado en *frameworks* para aplicaciones Web

Dependencias entre componentes del patrón



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?

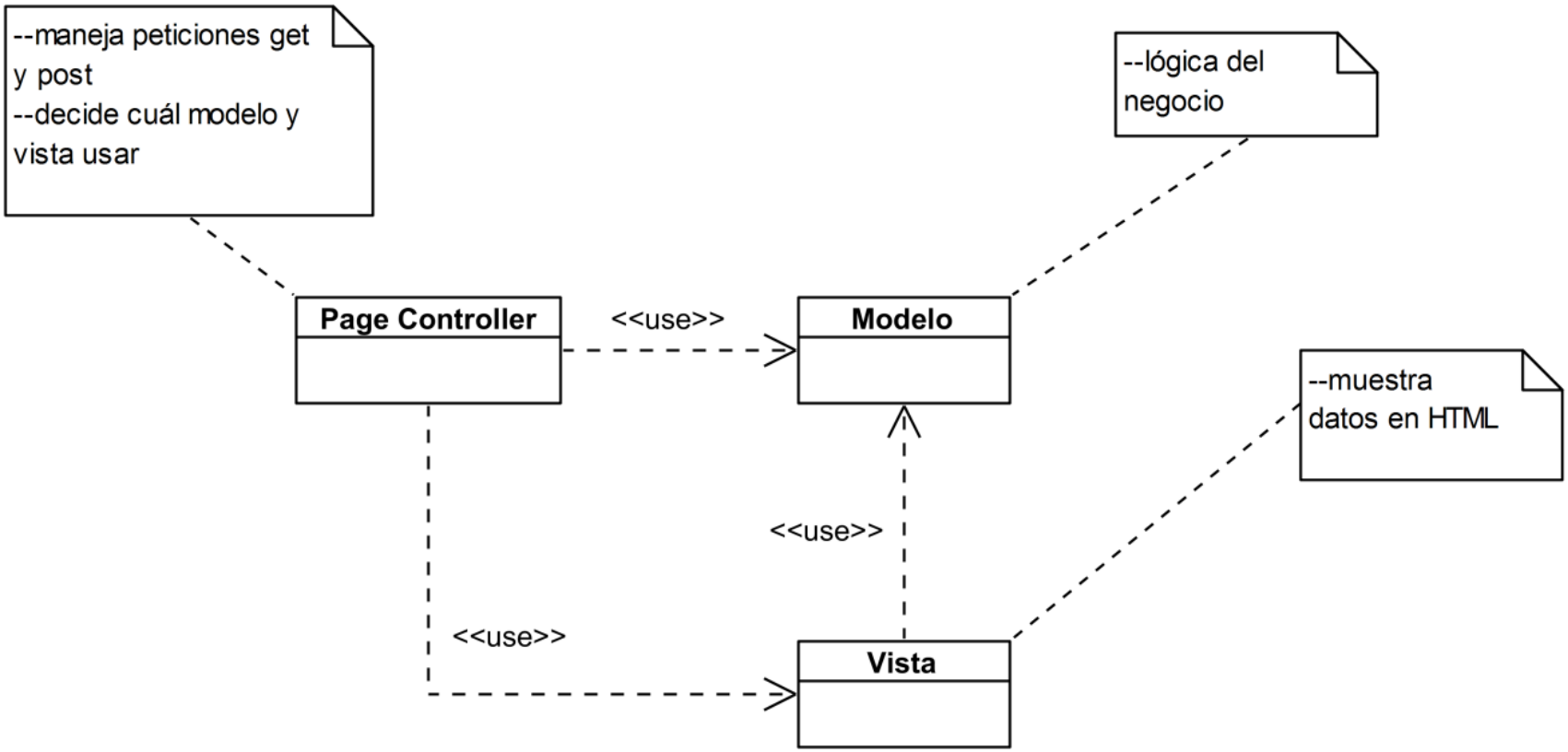
- MVC es un patrón de diseño que por sí solo es demasiado grande
- Cumple con la definición que hemos dado de un patrón de diseño…
  - Describe un problema recurrente
  - Entrega una propuesta de solución genérica aplicable a este problema recurrente
- Sin embargo, necesitamos más ayuda (de otros patrones)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

Page Controller

Page Controller [1]

- Ideas claves:
  - Un objeto se encarga de manejar las peticiones (requests) de una página (o acción) en un sitio Web
  - El objeto actúa como controlador de una página Web:
    - El controlador podría ser la página misma
    - O puede ser un objeto separado de la página
  - La implementación puede ser:
    - Un objeto por cada página
    - O un objeto por cada acción del usuario (ímplementación común)

Page Controller [2]



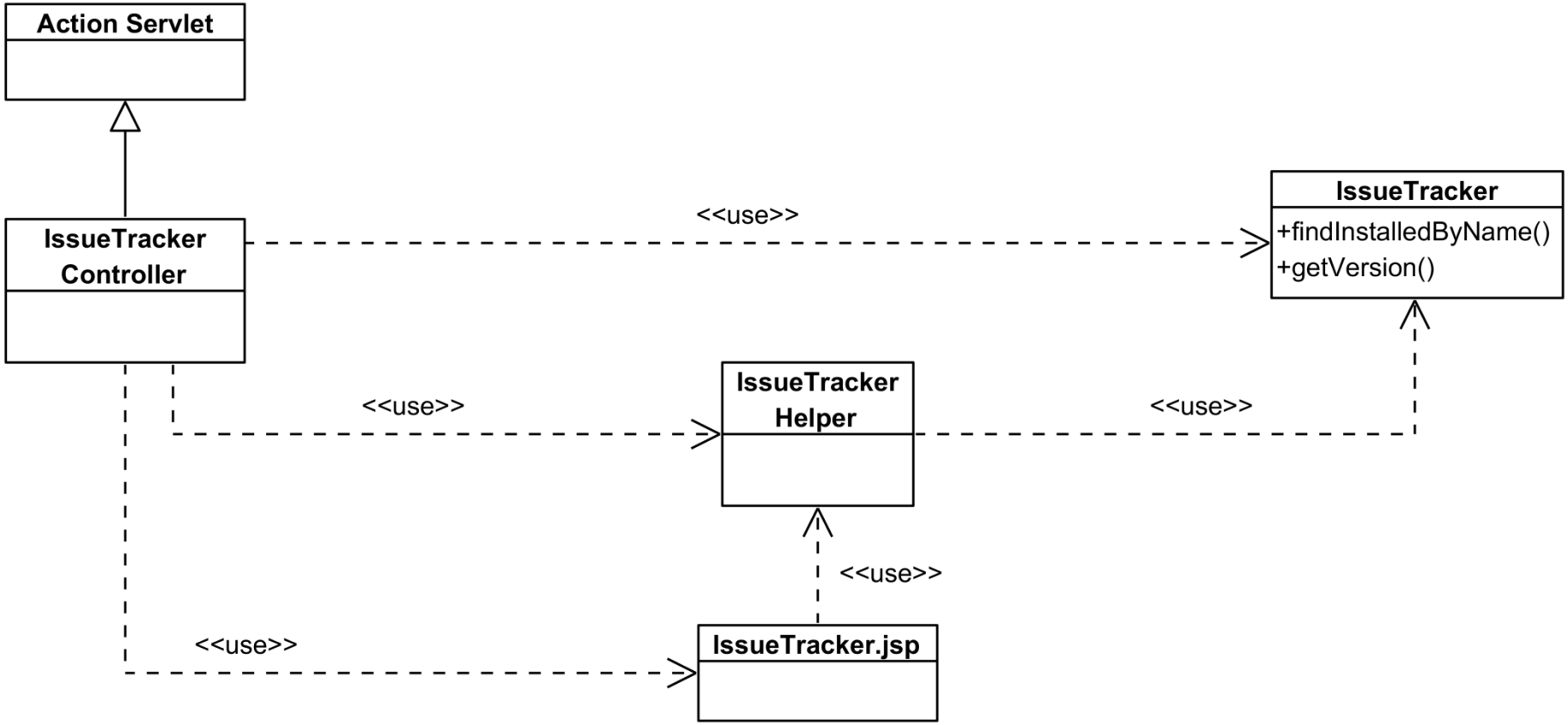
Page Controller: responsabilidades básicas

- Decodificar la URL/extraer datos desde formularios (incluye preparar los datos)
- Crear e invocar a los objetos que se encargarán de procesar los datos
- Determinar cuál vista debe desplegar el resultado y pasar los datos procesados a la misma

Servlets y JSP

- Servlets y JSP son dos tecnologías que sirven de base para construir ejemplos
- Un servlet es una clase JAVA que se utiliza para extender las capacidades de los servidores que mantienen aplicaciones
  - Utilizan el modelo “request-response”
  - Leen datos explícitos/implícitos enviados por el cliente
  - Envían datos explícitos/implícitos de vuelta al cliente
- JavaServer Pages (JSP) es una tecnología que permite crear páginas web con contenido estático y dinámico
  - Provee un lenguaje para la generación de contenido web
  - Provee mecanismos para acceder a objetos del lado del servidor

Ejemplo: JSP como Vista con un Controlador “Servlet” (JAVA) [1]



Template View

Template View

- Ideas claves:
  - Reproducir datos en HTML mediante el uso de "marcado"
  - Una página se compone de dos partes:
    - Parte estática
      - Actúa como *"template"*
    - Parte dinámica
      - El uso de "marcado" permite exponer datos
  - Las "marcas" son reemplazadas por los resultados de operaciones realizadas
    - Ej: Consulta a base de datos

Utilizando marcadores

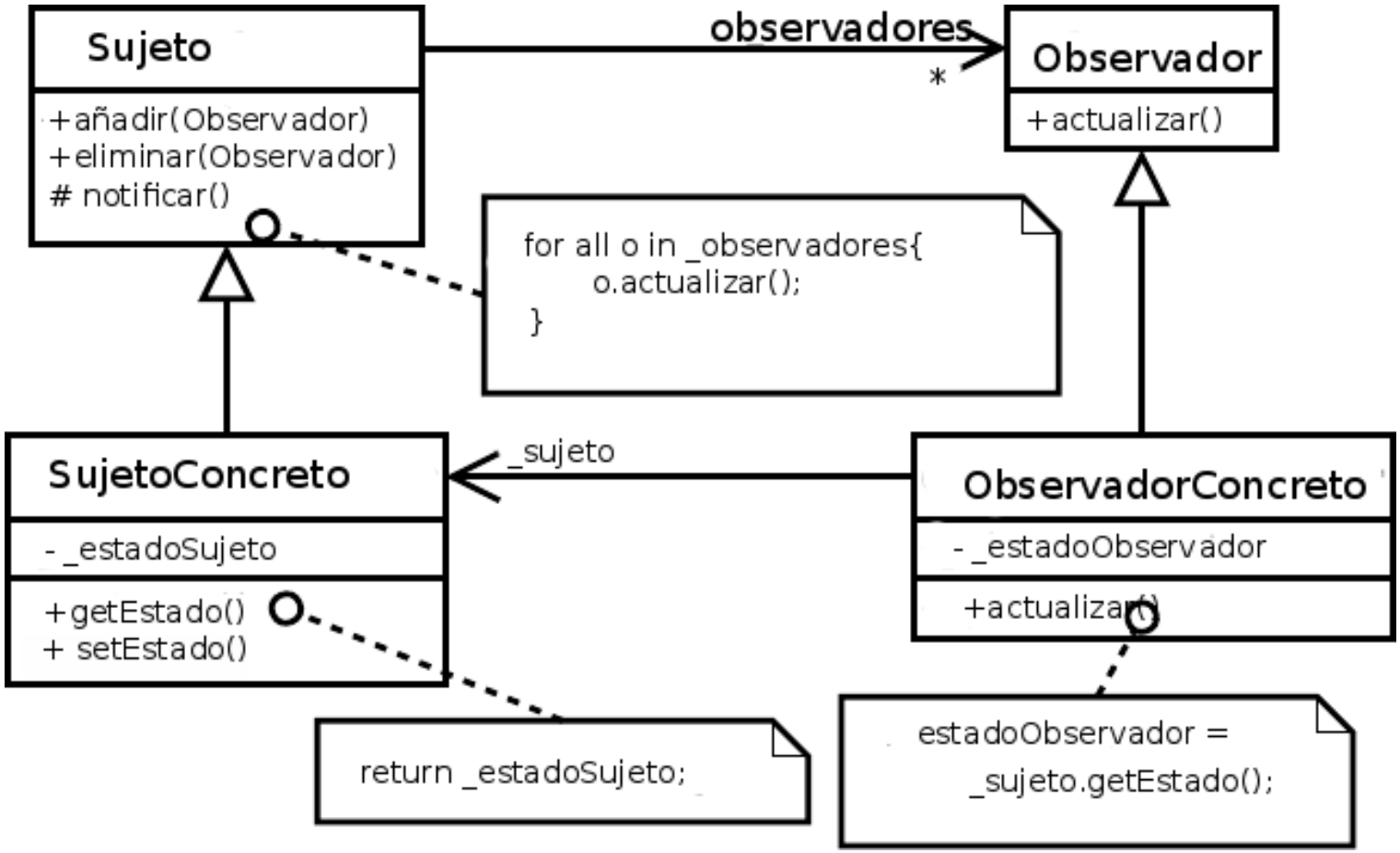
- Opción 1:
  - Aprovechar el formato de las etiquetas HTML
- Opción 2:
  - Marcadores especializados en el cuerpo del documento
- Ejemplos tradicionales:
  - “Server pages”: PHP, JSP, ASP
  - Riesgo: es tentador escribir código (lógica del negocio) en la página (“scriplets”)
    - Limita a los no-programadores (diseñadores, analistas de usabilidad, etc.) la posibilidad de editar la plantilla
    - Limita la modularidad del código

Desventajas

- Template View tiene dos desventajas significativas:
  - Permite la inclusión de código en la “plantilla”
    - Podemos terminar con un sistema lleno de “scriptlets”
  - Es difícil de *testear* dado que todo el código se ejecuta en el servidor

Transaction Script

Patrón Observador



Transaction Script

Transaction Script [1]

- Ideas claves:
  - Todo "negocio" tiene operaciones que lo representan y caracterizan
    - Ej: Comprar pasaje, Reservar habitación, Asignar Desarrollador a Proyecto
  - Las operaciones se expresan como transacciones en los sistemas de información que se encargan de interactuar con una base de datos y operar sobre estos datos
  - Ejemplo de transacción para "Comprar pasaje"
    1. Verificar disponibilidad
    2. Asignar pasaje a pasajero
    3. Registrar pago
    4. Cerrar/confirmar transacción *(commit)*

Transacción

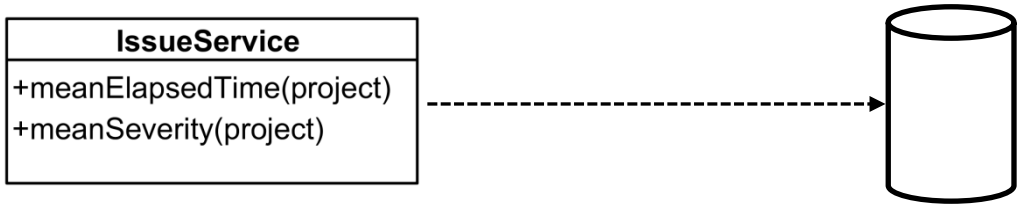
Transaction Script [2]

- Ideas claves:
  - La lógica del negocio se puede organizar en procedimientos (transacciones)
  - Cada procedimiento o transacción se encarga de manejar una solicitud singular desde la presentación
  - Podemos entonces crear un “Transaction Script” para cada uno de estos procedimientos

Transaction Script: Implementación

- Una forma común de utilizar este patrón consiste en agrupar un conjunto de transacciones en una clase
  - Requisito: la clase debe agrupar Transaction Scripts con una temática común
    - Por ejemplo, no podemos tener una clase "HotelService" con Transaction Scripts "BuyTicket" y "AssignDeveloperToProject"
- El acceso a la base de datos puede ser de dos formas:
  1. Separar la conexión y operación de los datos en otras clases, o
  2. Mantener todo el código de la conexión y operación de los datos en el mismo *script* (opción comúnmente usada)

Ejemplo: Transaction Script con JAVA y SQL [1]



Ejemplo: Transaction Script con JAVA y SQL [2]

```
class IssueService ... {
        public int meanElapsedTime (String projectName) {
                // obtener lista de issues asociados al proyecto
                // obtener, por cada uno de los issues, el tiempo
                // transcurrido (i.e., |closeDate – openDate|)
                // calcular promedio de los tiempos transcurridos
                return meanElapsedTime;
        }
        public int meanSeverity (String projectName) {
                // obtener lista de issues asociados al proyecto
                // obtener, por cada uno de los issues, la severidad
                // calcular el promedio de las severidades
                return meanSeverity;
        }
        // otras transacciones

}
```

Ejemplo: Transaction Script con JAVA y SQL [3]

- Comentarios:
  - Si bien el ejemplo es tradicional porque supone una base de datos SQL (relacional), nada impide que la base de datos sea de tipo NoSQL
  - El patrón sólo dice “encapsular acceso y operaciones sobre los datos”, no se especifica cómo se accede y cómo se opera sobre ellos

Transaction Script: Observaciones finales

- Beneficio:
  - Nos olvidamos de lo que otras transacciones hagan (una preocupación menos!)
  - Tiene un calce natural con la forma en que se entienden la mayoría de los negocios
  - Extremadamente simple de implementar!
- Desventajas:
  - TS no es una buena opción cuando la lógica del negocio es demasiado compleja
  - En sí, el patrón no establece que no se pueda llamar a la presentación (Vista) desde un TS
    - Pero esto no es una buena idea
      - Dificulta el testeo y el mantenimiento del sistema
      - Además, va en contra del esquema de dependencias entre los componentes de MVC

Domain Model

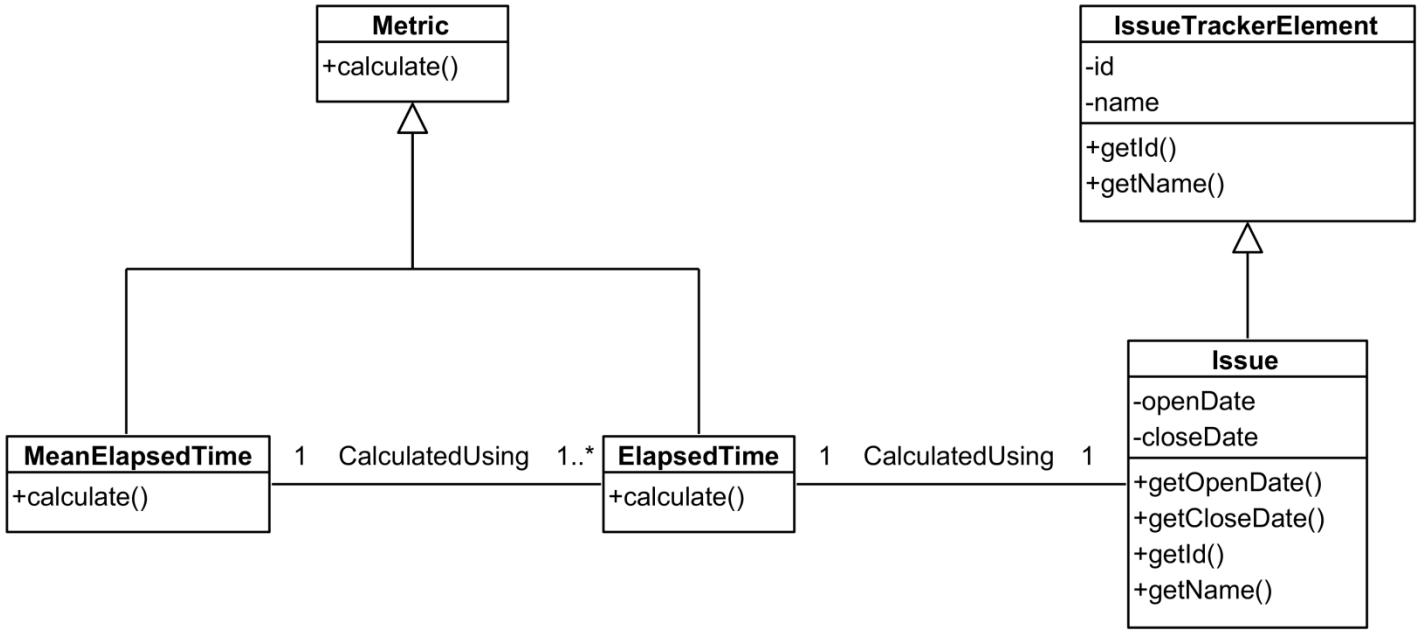
Domain Model

- Ideas claves:
  - Red de objetos interconectados que representan datos del negocio y capturan las reglas del negocio
  - Cada objeto tiene significado individual
    - Ej: Contrato, Persona, Pasaje, Issue, Venta
  - Caso simple: similar a un modelo relacional
  - Caso complejo: uso de herencia, composición, patrones, etc... (muy distante de un modelo relacional)
  - Sea cual sea el caso, no confundir un modelo de dominio con un modelo relacional
    - En esencia, es utilizar un modelo de dominio como los que ya conocemos, pero incorporando datos y comportamiento

Domain Model: Implementación

- Implica agregar una "capa de objetos" que modelan un área de negocio particular
- Mínimas dependencias entre el Modelo de Dominio y demás capas o partes del sistema
  - Así, cuando modificamos un objeto podemos estar más seguros de que no provocaremos problemas en otras partes
- La implementación es sencilla en una base de datos Orientada a Objetos
- Implementación usual involucra trabajo con bases de datos relacionales
  - Pero se puede implementar con cualquier tipo de base de datos

Ejemplo: Domain Model con Java y SQL [1]



Ejemplo: Domain Model con Java y SQL [2]

```
class Issue extends IssueTrackerElement {
        private int id;
        ...
        public String getOpenDate () {
                // SELECT openDate desde la base de datos
                // para un issue “id”
        }

        public String getCloseDate () {
                // SELECT closeDate desde la base de datos
                // para un issue “id”
        }
}
```

Ejemplo: Domain Model con Java y SQL [3]

```
class ElapsedTime extends Metric {
	public int calculate() {
		// obtener lista de ids de alguna forma
		Issue issue = new Issue(id);
		return (issue.getCloseDate() –
			issue.getOpenDate());
	}
}
class MeanElapsedTime extends Metric {
	public int calculate() {
		// crear lista con objetos ElapsedTime,
		// calcular promedio de los Elapsed Time
		// (que corresponden a un Issue particular)
		return meanElapsedTime;
	}
}
```

Domain Model: Observaciones finales

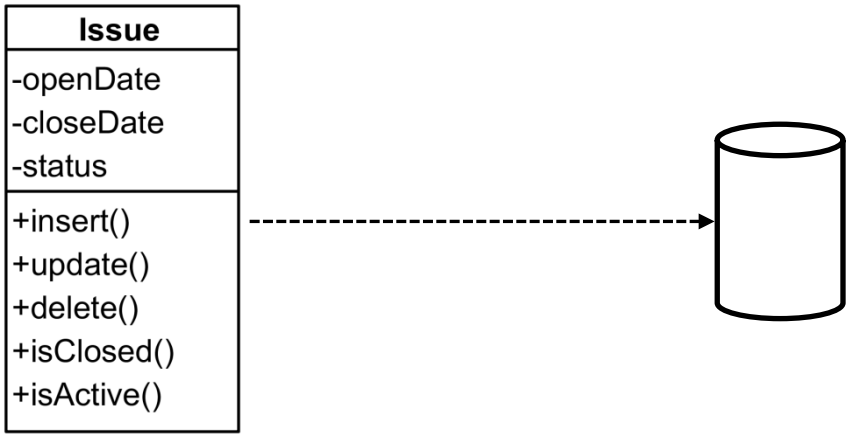
- La cantidad de objetos que se cargan en memoria es un tema a considerar
- Se sugiere mantener en memoria sólo los objetos necesarios
  - Por ejemplo, cargar sólo los Issues sobre los que se trabajará (ej: sólo Issues de un proyecto particular) y no todos los Issues de la base de datos)
- Es esperable que todas las clases tengan estado (datos) y comportamiento (métodos)
  - Aspecto importante: Domain Model es un patrón que hace uso intensivo del concepto de responsabilidad de los objetos

Active Record

Active Record [1]

- Ideas claves:
  - Un objeto “encapsula” una fila de una tabla de una base de datos, el acceso a la base de datos y provee lógica del dominio sobre los datos
  - En esencia, es un *Domain Model* que tiene dos responsabilidades:
    - Guardar y recuperar datos en la base de datos
    - Manejar los datos (lógica del dominio)
  - Se usa cuando las clases son muy similares a la estructuras de la base de datos

Ejemplo: Active Record con JAVA y SQL [1]



Ejemplo: Active Record con JAVA y SQL [2]

- Primero, tenemos la estructura de la base de datos:

```
CREATE TABLE issue (issueId INTEGER PRIMARY KEY,
                openDate DATE,
                        closeDate DATE,
                        status INTEGER, ...)
```

- Tenemos la clase que calza con la tabla:

```
class Issue ... {
        private int issueId;
        private String openDate;
        private String closeDate;
        private int status;
        ...
}
```

Ejemplo: Active Record con JAVA y SQL [3]

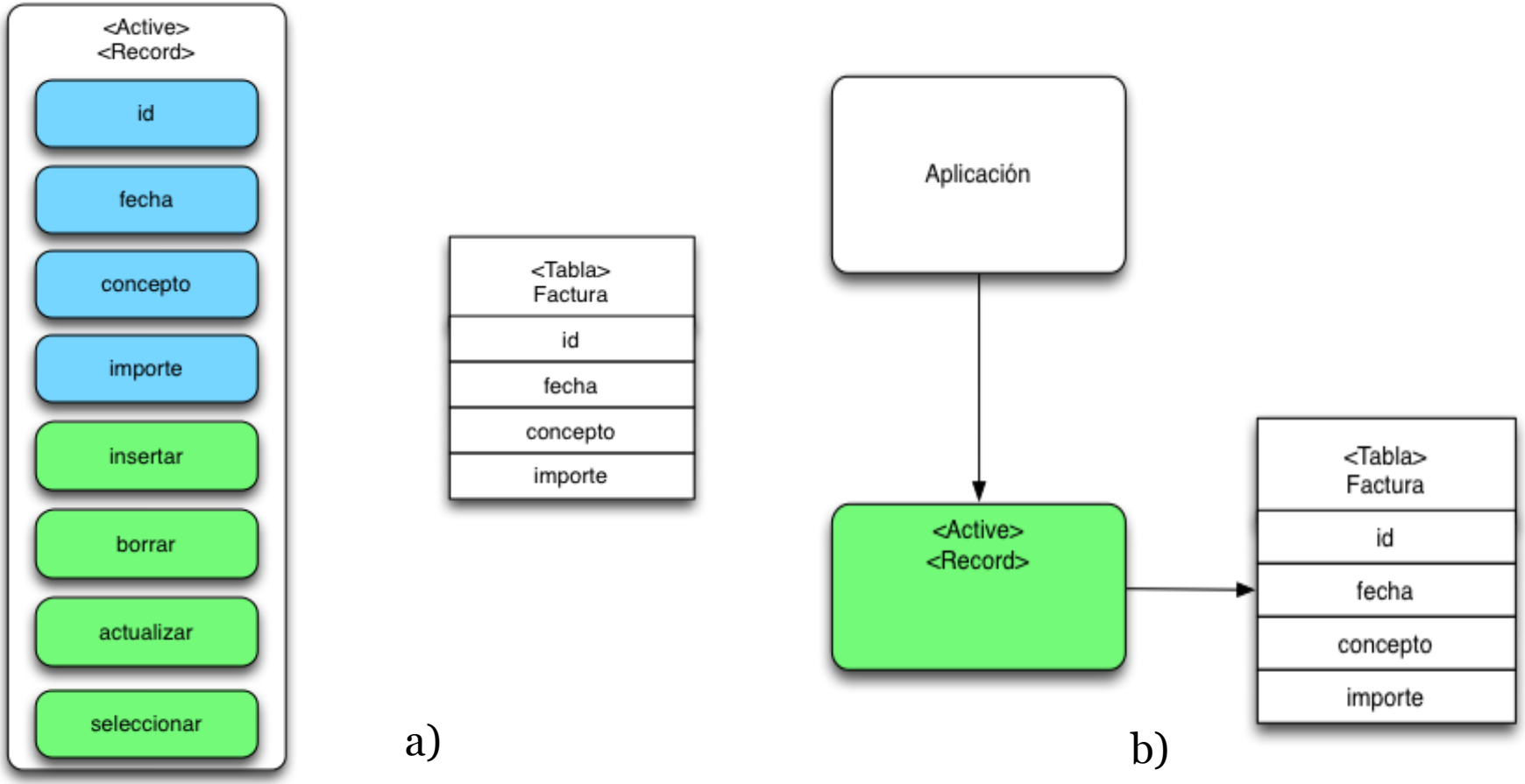
```
class Issue ...
      private int issueId;
      private String openDate;
      private String closeDate;
      private int status;

      public static Issue find (int id) {
            Issue issue = new Issue ();
            // código para buscar en SQL
            // ejecutar consulta de tipo SELECT
            // asignar valores de la forma:
            // issue.setStatus(RESULTADO SELECT)
            // (similar para demás atributos)
            return issue;
      }
```

Ejemplo: Active Record con JAVA y SQL [4]

```
class Issue ...
      // mismos atributos, continuación de la clase
      public void update () {
            // preparar sentencia para UPDATE con
            // datos que provienen del objeto Issue
            // WHERE issueId = getId()
            // ejecutar sentencia de actualización
      }
      public int insert () {
            setId (findNextTableID())
            // preparar sentencia para insertar en SQL
            // ejecutar sentencia para inserción
            return getId;
      }
```

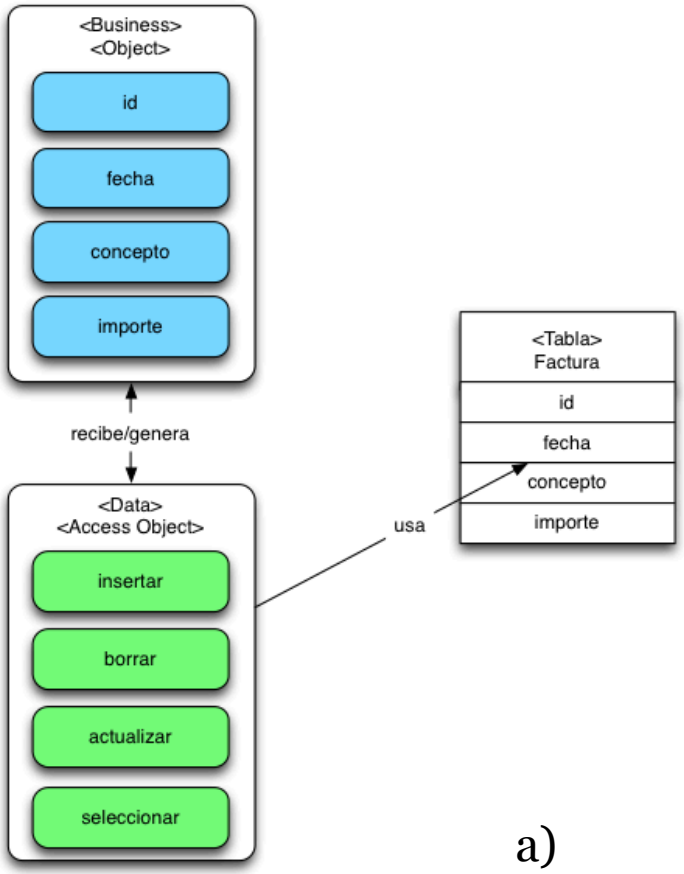
Active Record: Ejemplo



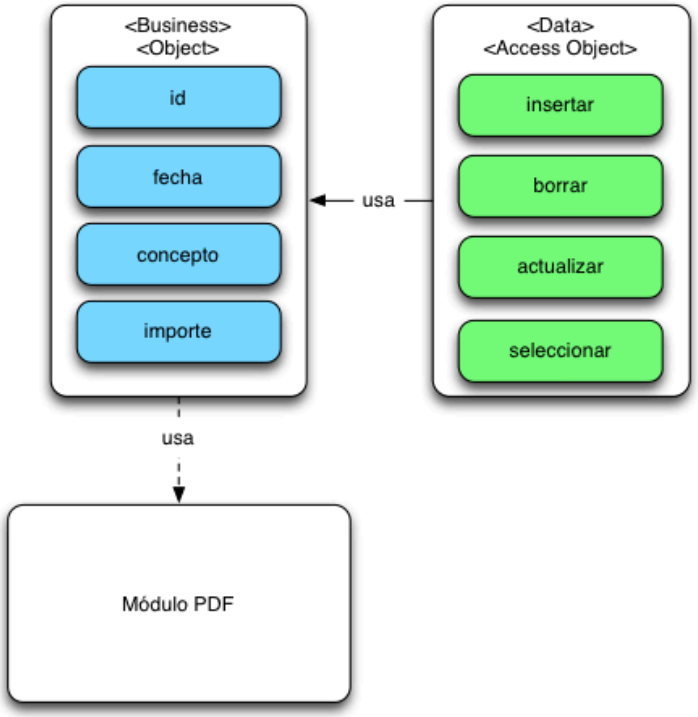
Active Record: Observaciones finales

- En el ejemplo vimos una tabla creada con SQL...
  - Pero nada impide que los datos sean almacenados en otro tipo de base de datos (incluso NoSQL)
- Active Record funciona bien cuando las operaciones son de tipo CREATE, INSERT, DELETE, UPDATE y otras similares
  - Cuando la complejidad de las operaciones aumenta, Active Record puede volverse difícil de usar
- Active Record se acopla fuertemente al diseño de la base de datos
  - Se puede hacer difícil el *refactoring* del código

DAO


<Business>
<Object>
id
fecha
concepto
importe
recibe/genera
<Data>
<Access Object>
insertar
borrar
actualizar
seleccionar
usa
<Tabla>
Factura
id
fecha
concepto
importe

a)


<Business>
<Object>
id
fecha
concepto
importe
usa
<Data>
<Access Object>
insertar
borrar
actualizar
seleccionar
usa
Módulo PDF

b)

MVC: Comentarios finales

MVC: Comentarios finales

- La esencia en MVC es la separación de componentes en un sistema
- MVC no especifica cómo se debe implementar en forma específica la separación
- Lo importante en MVC es cumplir con las dependencias entre los componentes...
- ... y cumplir con la separación entre las capas (componentes)
- Todos los patrones que hemos visto (Page Controller, Template View, Transaction Script, Domain Model y Active Record) se pueden utilizar para construir sistemas MVC

FIN