Arquitectura de la capa de presentación

ISW – Ingeniería de Software para la Web

Wladimiro Díaz

Departament d'Informàtica
Escola Técnica Superior d'Enginyeria
Universitat de València

Máster Ingeniería de Sevicios y Aplicaciones Web 2013-14

Indice

1 Introducción

2 El patrón Modelo-Vista-Controlador
- El patrón MVC en J2EE
- El patrón factory
- El patrón Front Controller
- El patrón Intercepting Filter

3 Diseño avanzado de la capa de presentación

Indice

1. Introducción

2. El patrón Modelo-Vista-Controlador
   - El patrón MVC en J2EE
   - El patrón factory
   - El patrón Front Controller
   - El patrón Intercepting Filter

3. Diseño avanzado de la capa de presentación

Indice

1. Introducción

2. El patrón Modelo-Vista-Controlador
   - El patrón MVC en J2EE
   - El patrón factory
   - El patrón Front Controller
   - El patrón Intercepting Filter

3. Diseño avanzado de la capa de presentación

Indice

1 Introducción

2 El patrón Modelo-Vista-Controlador

3 Diseño avanzado de la capa de presentación

¿Cómo gestionar la evolución del software?

- No hay una respuesta simple.
- Sin embargo, para mantener un producto en desarrollos a largo plazo, es necesario disponer de una arquitectura que:
  - que permita al programador extender y reorganizar los diferentes componentes.
  - que balancee adecuadamente las características de flexibilidad, extensibilidad y rendimiento.
- En J2EE, la capa de presentación es responsable de una gran parte de la complejidad de la aplicación.
- Además, es la principal responsable de la “experiencia de usuario”, por lo que es un candidato permanente al cambio.
- La capa de presentación se construye fundamentalmente con los servlets y las páginas HTML y JSP.

Objetivo

- Si ya que sabemos que el software evoluciona, debemos afrontar el desarrollo planificando el cambio.
- Es imprescindible introducir la extensibilidad en la arquitectura subyacente.
- A continuación analizaremos los patrones que afectan al diseño global de la capa de presentación:
  - **Modelo-Vista-Controlador**. Proporciona una arquitectura para la totalidad de la capa de presentación que separa claramente el estado, la presentación y el comportamiento.
  - **Front-Controlle**. Centraliza el acceso en un entorno basado en peticiones.
  - **Decorator**. Permite añadir dinámicamente funcionalidad a un controlador.

Indice

1 Introducción

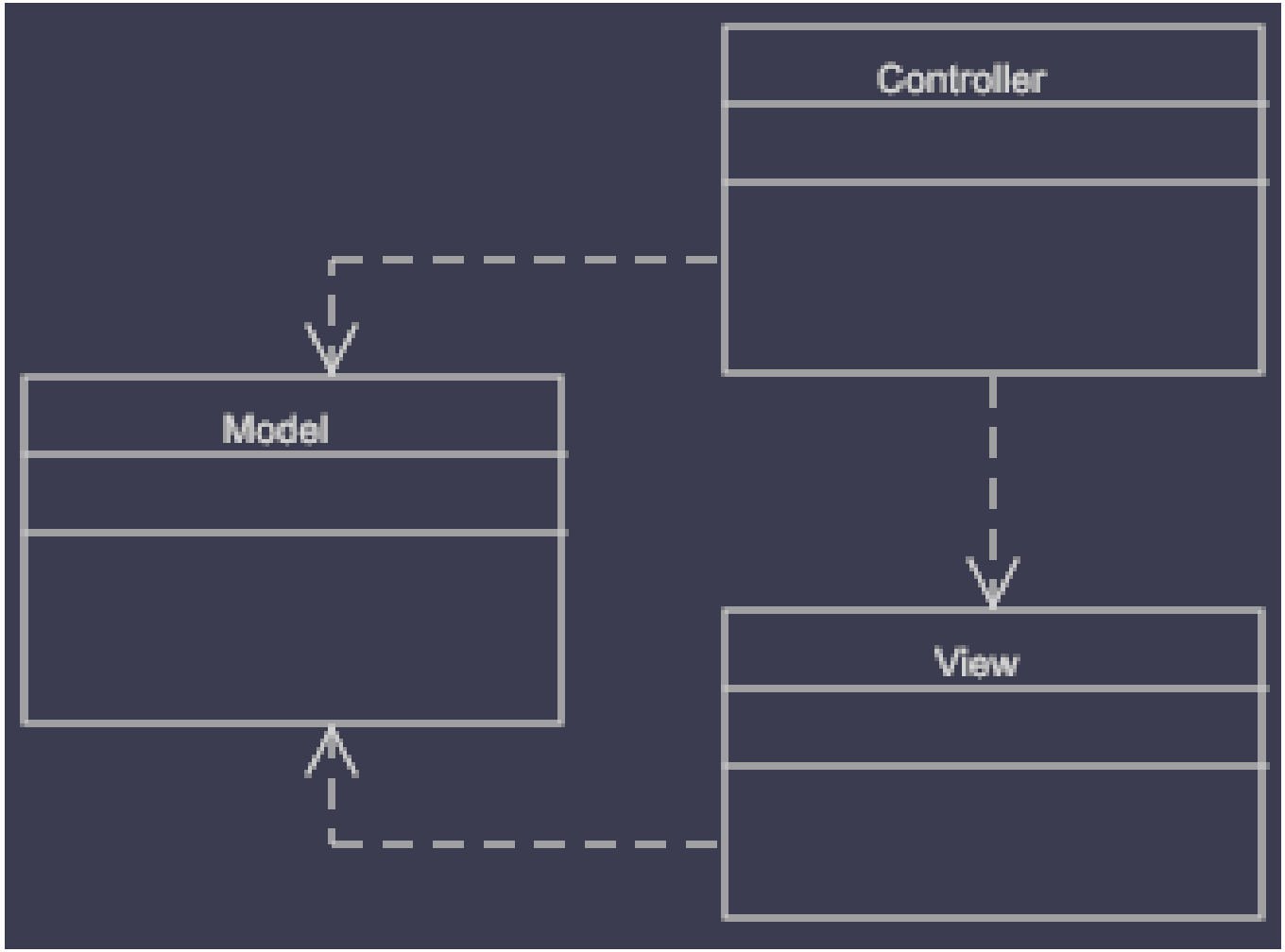
2 El patrón Modelo-Vista-Controlador
- El patrón MVC en J2EE
- El patrón factory
- El patrón Front Controller
- El patrón Intercepting Filter

3 Diseño avanzado de la capa de presentación

El patrón MVC

- Como sugiere su nombre, el patrón MVC descompone la interface de usuario en tres pieces distintas: modelo, vista y controlador.
  - El modelo almacena el estado de la aplicación.
  - La vista (o vistas) interpreta los datos del modelo y los presenta al usuario.
  - Por último, el controlador procesa las acciones del usuario, actualizando el modelo y/o mostrando una nueva vista.
- La división de tareas y un control adecuado de las comunicaciones entre estas tres piezas permite construir la arquitectura robusta necesaria.

Diagrama de clases del patrón MVC



Una aplicación de ejemplo

- Supongamos una página web que permite a un usuario registrarse en una lista de distribución.
- El usuario introduce el nombre, el apellido y la dirección de correo electrónico en un formulario.
- Cuando se pulsa `Submit`, la aplicación añade su dirección de correo a una lista e informa del resultado de la operación.
- Puesto que el formulario no cambia, lo crearemos como una página HTML convencional.

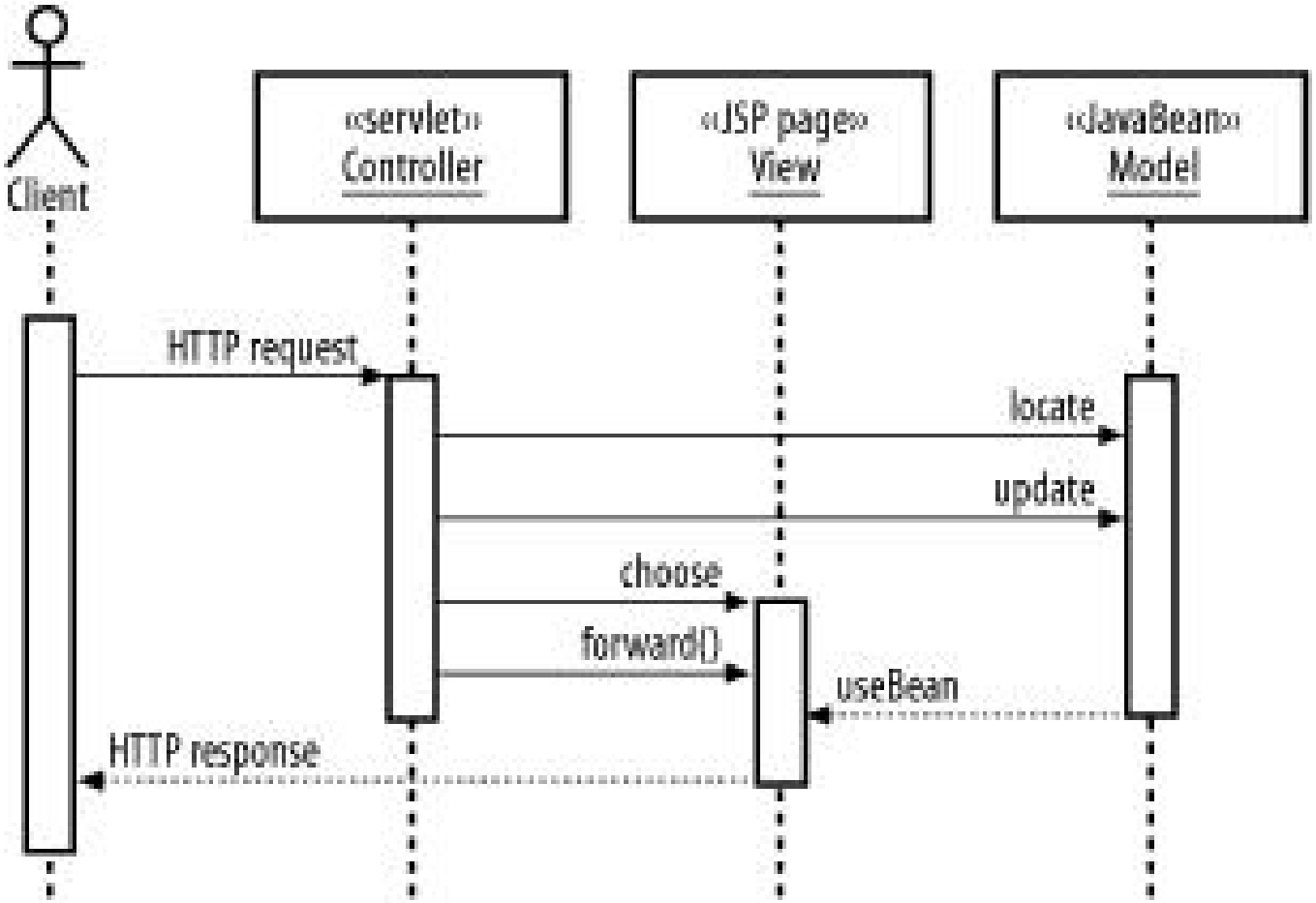
El formulario de datos

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
2    "http :// www.w3.org/TR/html4/loose.dtd">
3  <html>
4    <head>
5      <meta http-equiv="Content-Type"
6          content="text/html; charset=UTF-8">
7      <title>Subscribe!</title>
8    </head>
9    <body>
10     <form action="ListController" method="get">
11       First  Name: <input type="text" name="first"> <br>
12       Last Name: <input type="text" name="last"> <br>
13       Email Address: <input type="text" name="email"> <br>
14       <input type="submit" name="Subscribe!">
15     </form>
16   </body>
17 </html>
```

El Bean

```
1  public class MailingBean {
2    private String first = "", last = "", email = "";
3    public String getFirst() {
4      return first;
5    }
6    public void setFirst(String first) {
7      this.first = first;
8    }
9    public String getLast() {
10     ...
11     ...
12   public boolean doSubscribe() {
13     if (email.contains("@"))
14       return true;
15     else
16       return false;
17   }
18   public String getErrorString() {
19     return "This string doesn't seem to represent a valid e-mail address";
20   }
21 }
```

Interacciones MVC en J2EE



El patrón factory

- **Características**:
  - Probablemente el patrón de diseño más usado en Java y otros lenguajes.
  - Cuenta con multitud de variantes e implementaciones.
  - En los patrones GoF se puede encontrar como **Factory Method** y **Abstract Factory**.
  - Aquí veremos la forma simple y las implementaciones avanzadas en una asignatura posterior.
- **Objetivo**:
  - Crear objetos ocultando la lógica de instanciación al cliente.
  - El cliente hace referencia al objeto recién creado a través de una **interface** común.

El patrón factory

- **Características**:
  - Probablemente el patrón de diseño más usado en Java y otros lenguajes.
  - Cuenta con multitud de variantes e implementaciones.
  - En los patrones GoF se puede encontrar como **Factory Method** y **Abstract Factory**.
  - Aquí veremos la forma simple y las implementaciones avanzadas en una asignatura posterior.
- **Objetivo**:
  - Crear objetos ocultando la lógica de instanciación al cliente.
  - El cliente hace referencia al objeto recién creado a través de una **interface** común.

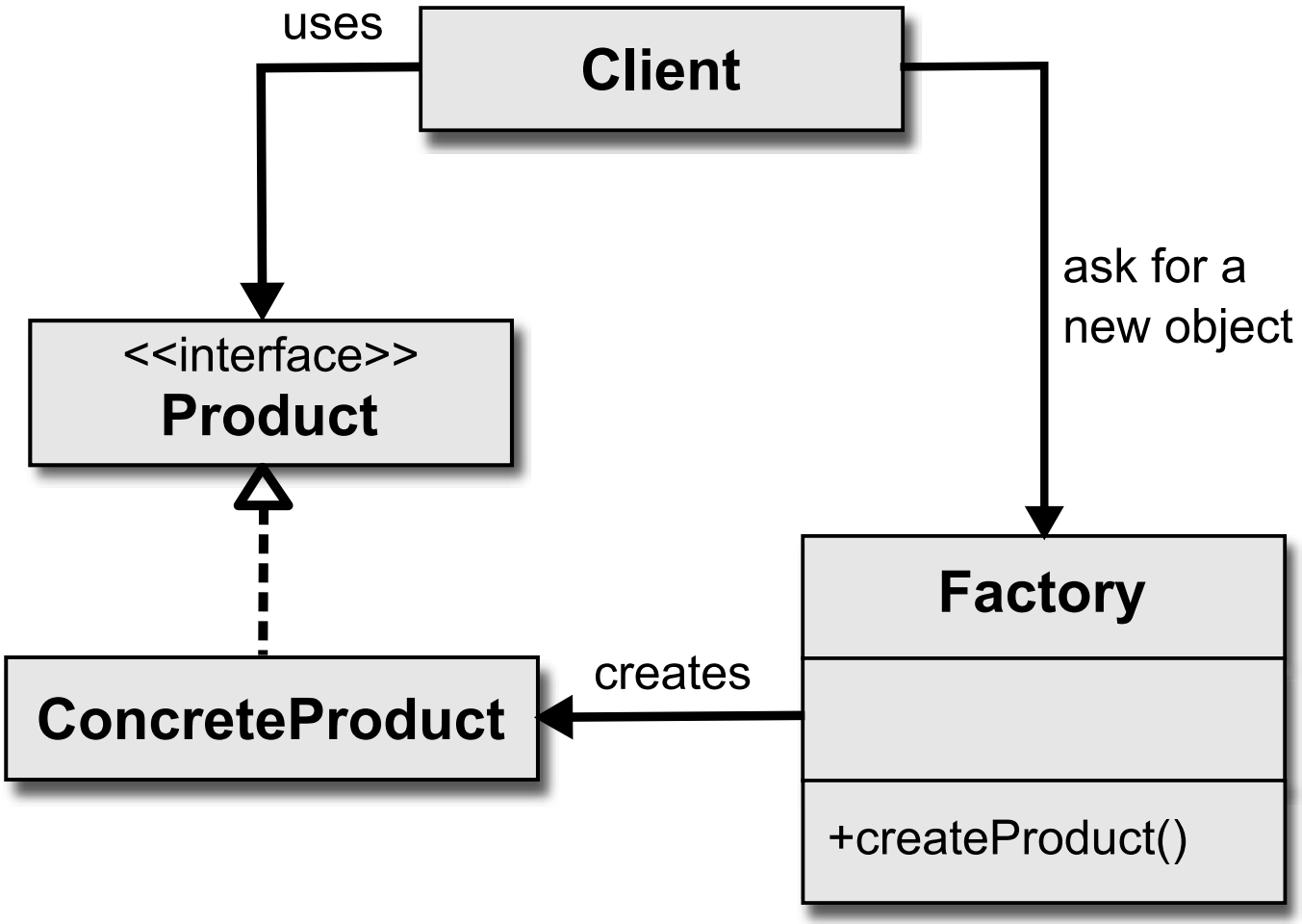
El patrón factory

- **Características**:
  - Probablemente el patrón de diseño más usado en Java y otros lenguajes.
  - Cuenta con multitud de variantes e implementaciones.
  - En los patrones GoF se puede encontrar como **Factory Method** y **Abstract Factory**.
  - Aquí veremos la forma simple y las implementaciones avanzadas en una asignatura posterior.
- **Objetivo**:
  - Crear objetos ocultando la lógica de instanciación al cliente.
  - El cliente hace referencia al objeto recién creado a través de una **interface** común.

Implementación

- El cliente necesita un producto, pero no lo crea directamente mediante el comando **new**.
- En lugar de ello, solicita el nuevo producto a la factoria indicando qué tipo de objeto necesita.
- La factoría instancia un producto concreto y se lo devuelve al cliente (depués de hacer un *cast* a la clase del producto abstracto.
- El cliente usa el producto abstracto sin necesidad de gestionar los detalles de la implementación concreta.

Implementación



Aplicabilidad

```
1  package es.uv.isw.app1.beans;
2
3  public interface MailingBean {
4     //  First  name
5     public String  getFirst ();
6     public void  setFirst (String   first  );
7
8     //  Last name
9     public String  getLast ();
10    public void setLast(String  last );
11
12    //  email address
13    public String  getEmail();
14    public void setEmail(String email);
15
16    //  business method
17    public boolean doSubscribe();
18
19    //  subscription  result
20    public String  getErrorString ();
21 }
```

En nuevo bean

```
1  package es.uv.isw.app1.beans;
2  public class MailingBeanImpl implements MailingBean {
3    private String first = "", last = "", email = "";
4    @Override
5    public String getFirst () {
6      return first ;
7    }
8     ...
9     ...
10   @Override
11   public boolean doSubscribe() {
12     if (email.contains("@"))
13       return true;
14     else
15       return false;
16   }
17   @Override
18   public String getErrorString () {
19     return "This string doesn't seem to represent a valid e-mail address";
20   }
21 }
```

La factoría

```
1 package es.uv.isw.app1.beans;
2 public class MailingBeanFactory {
3    public static  MailingBean newInstance() {
4       return new MailingBeanImpl();
5    }
6 }
```

Ventajas

La ventaja es obvia: es posible añadir nuevas implementaciones del bean, adaptadas a las necesidades de la aplicación, simplemente modificando una línea de código en la factoría.

La factoría

```
1 package es.uv.isw.app1.beans;
2 public class MailingBeanFactory {
3    public static  MailingBean newInstance() {
4       return new MailingBeanImpl();
5    }
6 }
```

Ventajas

La ventaja es obvia: es posible añadir nuevas implementaciones del bean, adaptadas a las necesidades de la aplicación, simplemente modificando una línea de código en la factoría.

Diagrama de clases

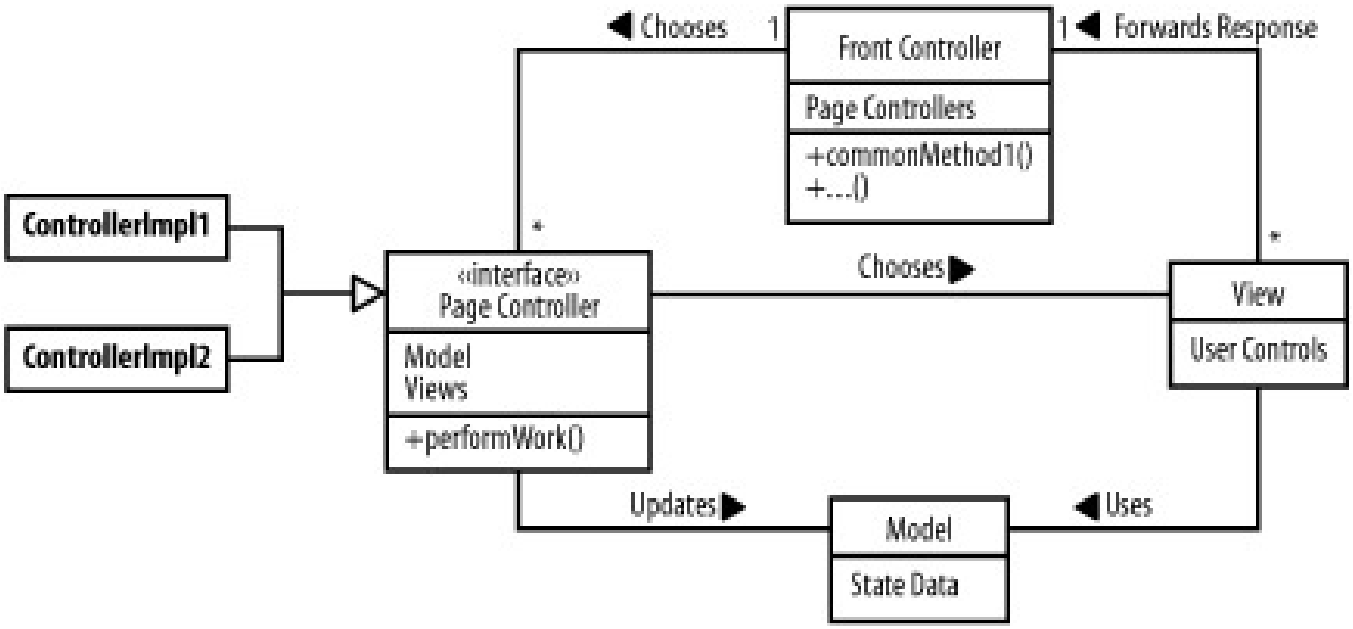
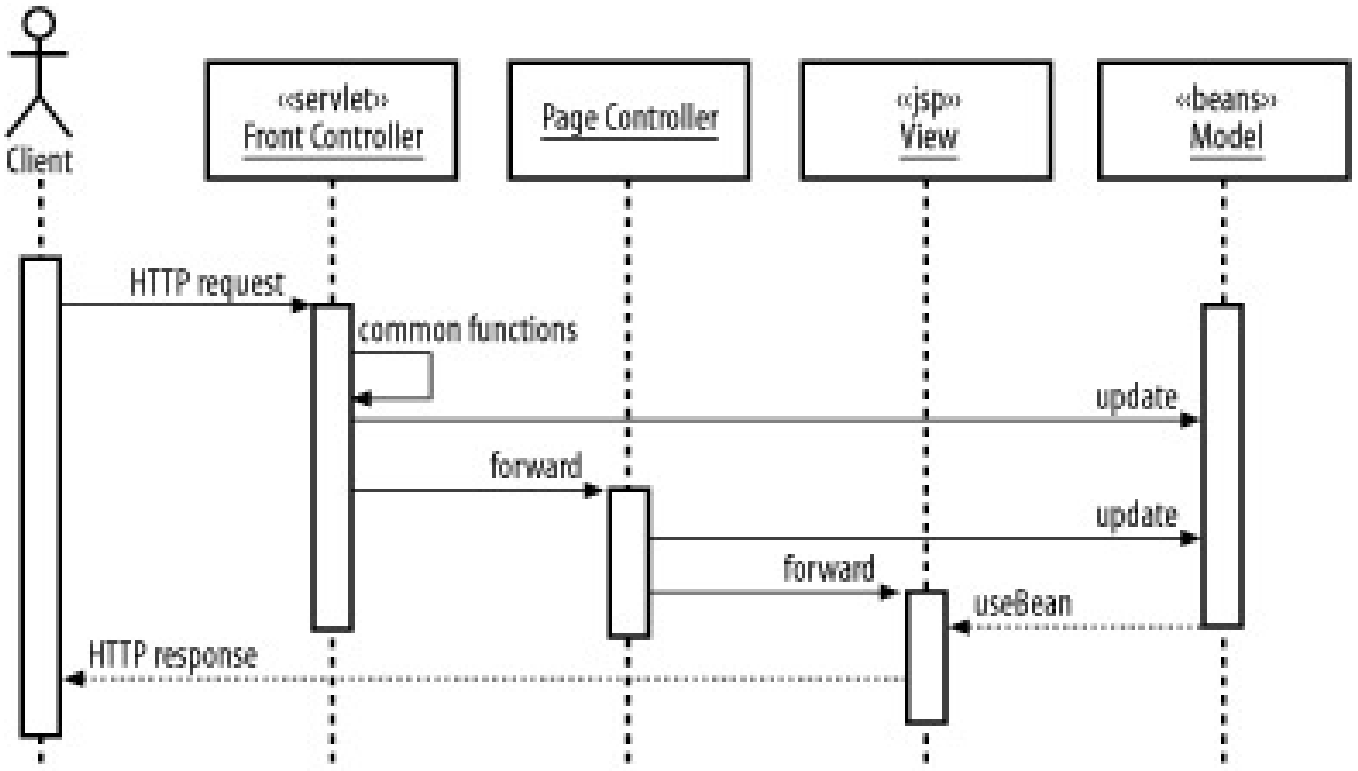


Diagrama de secuencia en J2EE



El contexto

- La capa de presentación recibe peticiones de múltiples tipos, que requieren un procesamiento específico.
  - Para algunas peticiones es suficiente con redireccionar la petición al componente adecuado.
  - Sin embargo, para otras peticiones es necesario realizar algún tipo de procesamiento adicional, tales como modificaciones, auditoría, etc...

El problema

El problema

Preprocesar y postprocesar las peticiones de un cliente web

- Es frecuente que cuando una aplicación Web recive una petición, sea necesario realizar una serie de pruebas antes de la etapa principal de procesado.
- Ejemplos:
  - ¿Está el cliente autenticado?
  - ¿Es la sesión del cliente válida?
  - ¿Pertenece la IP del cliente a una red autorizada o fiable?
  - ¿Está soportado el navegador del cliente?
  - ¿Es el *path* de la *request* válido?
- Algunos de estas comprobaciones son de tipo *true/false* determinando si el procesamiento debe continuar o no.
- Otras manipulan los datos de entrada para que se puedan procesar adecuadamente.

El problema

El problema

Preprocesar y postprocesar las peticiones de un cliente web

- Es frecuente que cuando una aplicación Web recive una petición, sea necesario realizar una serie de pruebas antes de la etapa principal de procesado.
- Ejemplos:
  - ¿Está el cliente autenticado?
  - ¿Es la sesión del cliente válida?
  - ¿Pertenece la IP del cliente a una red autorizada o fiable?
  - ¿Está soportado el navegador del cliente?
  - ¿Es el *path* de la *request* válido?
- Algunos de estas comprobaciones son de tipo *true/false* determinando si el procesamiento debe continuar o no.
- Otras manipulan los datos de entrada para que se puedan procesar adecuadamente.

El problema

El problema

Preprocesar y postprocesar las peticiones de un cliente web

- Es frecuente que cuando una aplicación Web recive una petición, sea necesario realizar una serie de pruebas antes de la etapa principal de procesado.
- Ejemplos:
  - ¿Está el cliente autenticado?
  - ¿Es la sesión del cliente válida?
  - ¿Pertenece la IP del cliente a una red autorizada o fiable?
  - ¿Está soportado el navegador del cliente?
  - ¿Es el *path* de la *request* válido?
- Algunos de estas comprobaciones son de tipo *true/false* determinando si el procesamiento debe continuar o no.
- Otras manipulan los datos de entrada para que se puedan procesar adecuadamente.

El problema

El problema

Preprocesar y postprocesar las peticiones de un cliente web

- Es frecuente que cuando una aplicación Web recive una petición, sea necesario realizar una serie de pruebas antes de la etapa principal de procesado.
- Ejemplos:
  - ¿Está el cliente autenticado?
  - ¿Es la sesión del cliente válida?
  - ¿Pertenece la IP del cliente a una red autorizada o fiable?
  - ¿Está soportado el navegador del cliente?
  - ¿Es el *path* de la *request* válido?
- Algunos de estas comprobaciones son de tipo *true/false* determinando si el procesamiento debe continuar o no.
- Otras manipulan los datos de entrada para que se puedan procesar adecuadamente.

El problema

- La solución clásica consiste en una serie de bloques condicionales (sentencias `if/else`). Si alguna de las comprobaciones falla, se aborta la consulta.
- Sin embargo, esta solución conduce a código frágil y a una programación de tipo copy&paste.
- El flujo de las operaciones de filtrado y la acción de los filtros se compilan dentro de la aplicación.

La solución

Diseñar componentes de proceso que completen acciones específicas de filtrado y disponer de un mecanismo simple, flexible y sin efectos colaterales para añadir y eliminar estos componentes.

El problema

- La solución clásica consiste en una serie de bloques condicionales (sentencias `if/else`). Si alguna de las comprobaciones falla, se aborta la consulta.
- Sin embargo, esta solución conduce a código frágil y a una programación de tipo copy&paste.
- El flujo de las operaciones de filtrado y la acción de los filtros se compilan dentro de la aplicación.

La solución

Diseñar componentes de proceso que completen acciones específicas de filtrado y disponer de un mecanismo simple, flexible y sin efectos colaterales para añadir y eliminar estos componentes.

El problema

- La solución clásica consiste en una serie de bloques condicionales (sentencias `if/else`). Si alguna de las comprobaciones falla, se aborta la consulta.
- Sin embargo, esta solución conduce a código frágil y a una programación de tipo copy&paste.
- El flujo de las operaciones de filtrado y la acción de los filtros se compilan dentro de la aplicación.

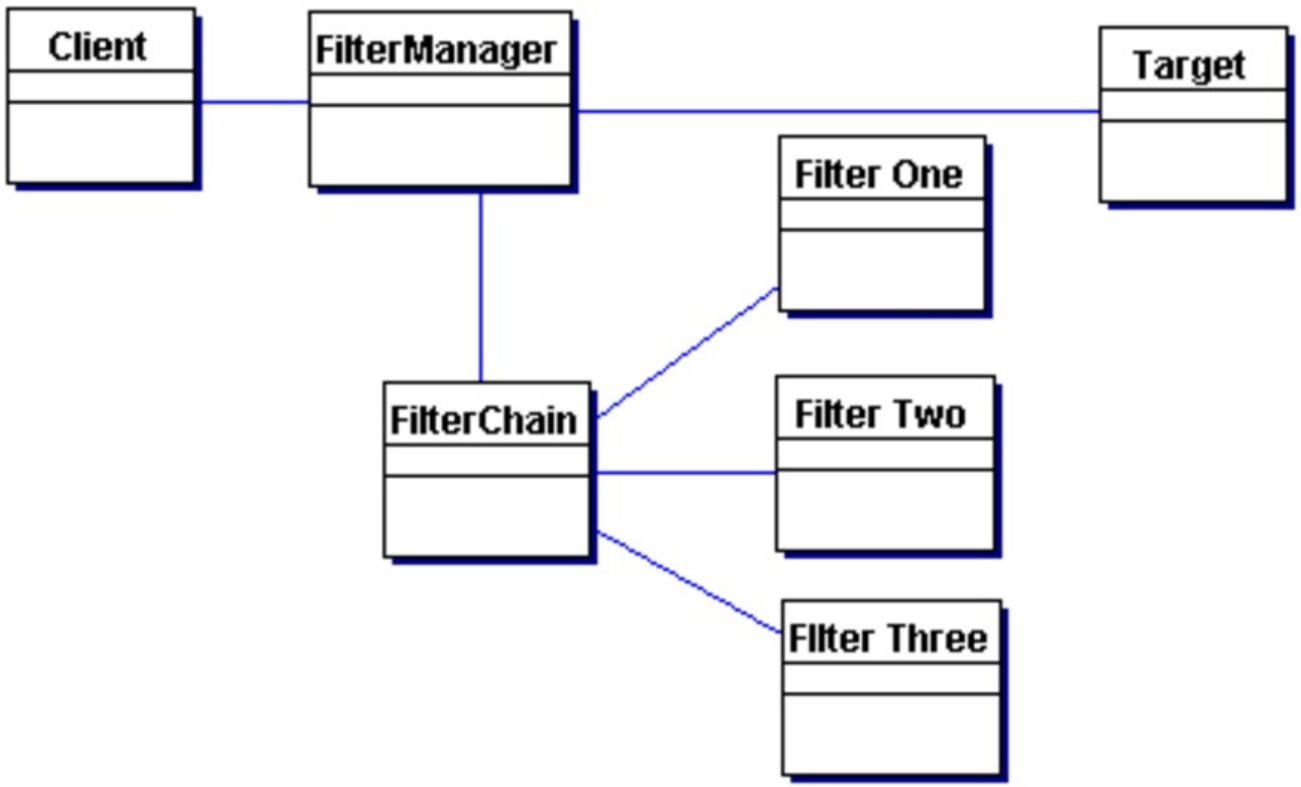
La solución

Diseñar componentes de proceso que completen acciones específicas de filtrado y disponer de un mecanismo simple, flexible y sin efectos colaterales para añadir y eliminar estos componentes.

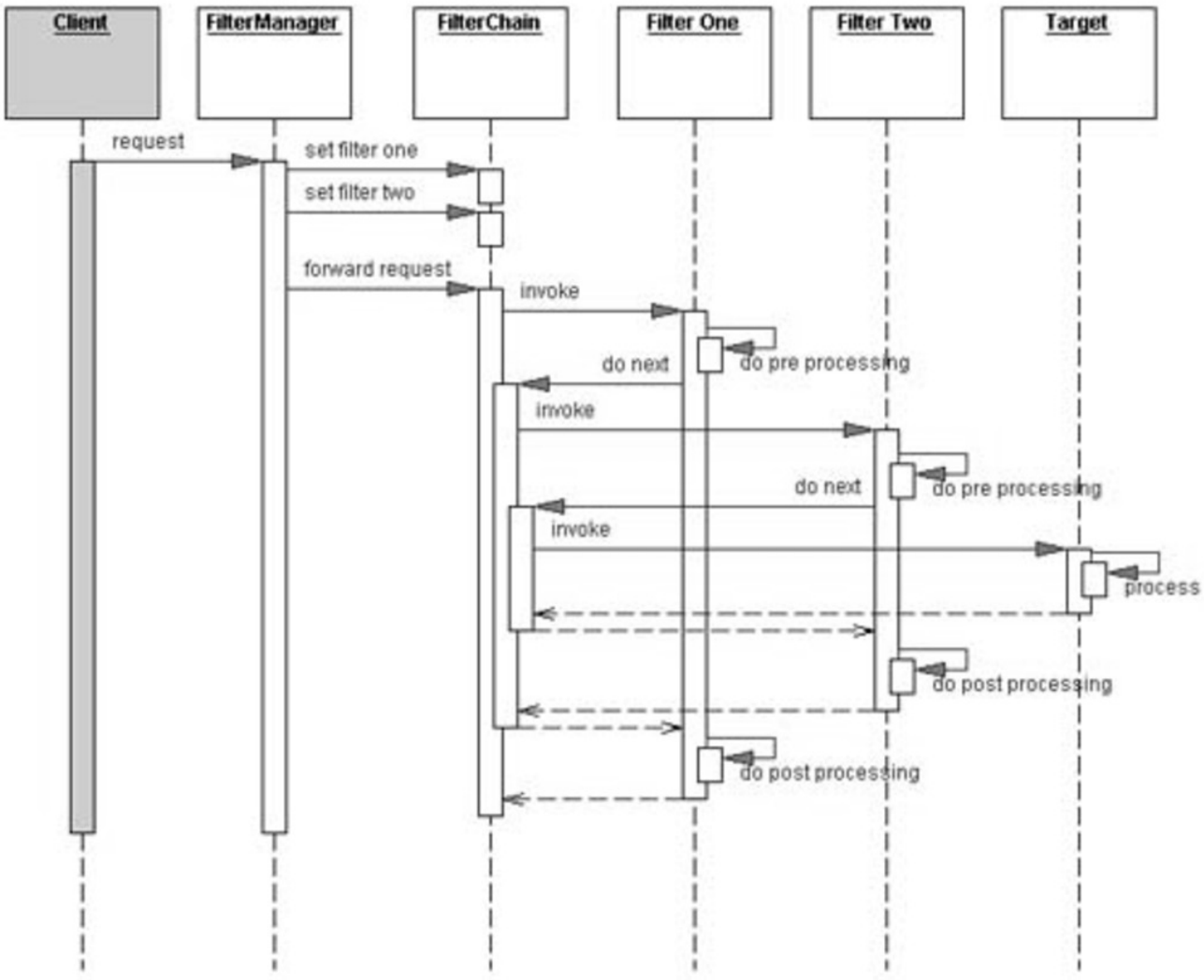
La solución

- Crear filtros **pluggable** para procesar servicios comunes de forma estándar que no requieran modificar el código básico de procesamiento de la petición.
- El filtro intercepta la petición y la respuesta, permitiendo su **pre-procesado** y el **post-procesado**.
- Estos filtros deben ser añadidos y eliminados de forma no obstructiva y sin la necesidad de realizar modificaciones en el código existente.

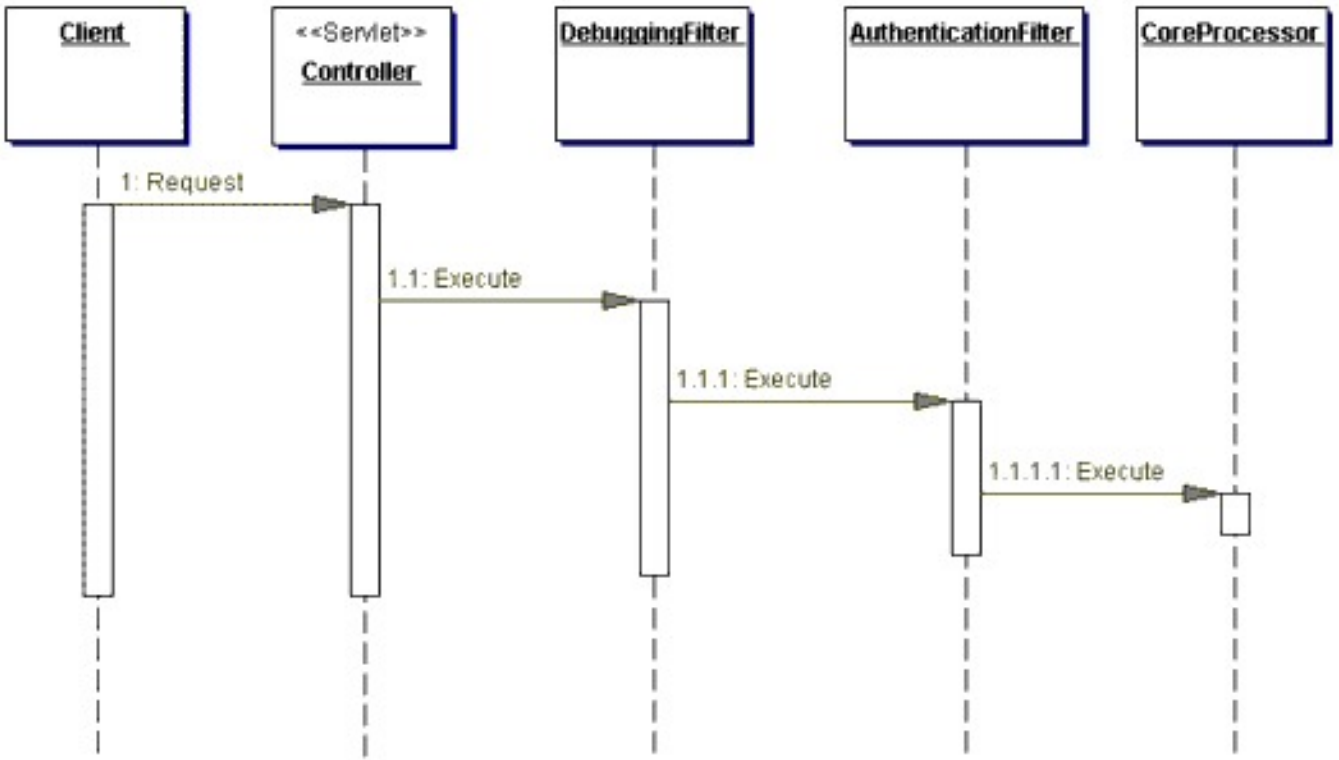
Estructura


Client
FilterManager
Target
Filter One
FilterChain
Filter Two
Filter Three

Participantes



La práctica


Client
<<Servlet>>
Controller
DebuggingFilter
AuthenticationFilter
CoreProcessor
1: Request
1.1: Execute
1.1.1: Execute
1.1.1.1: Execute

Indice

1 Introducción

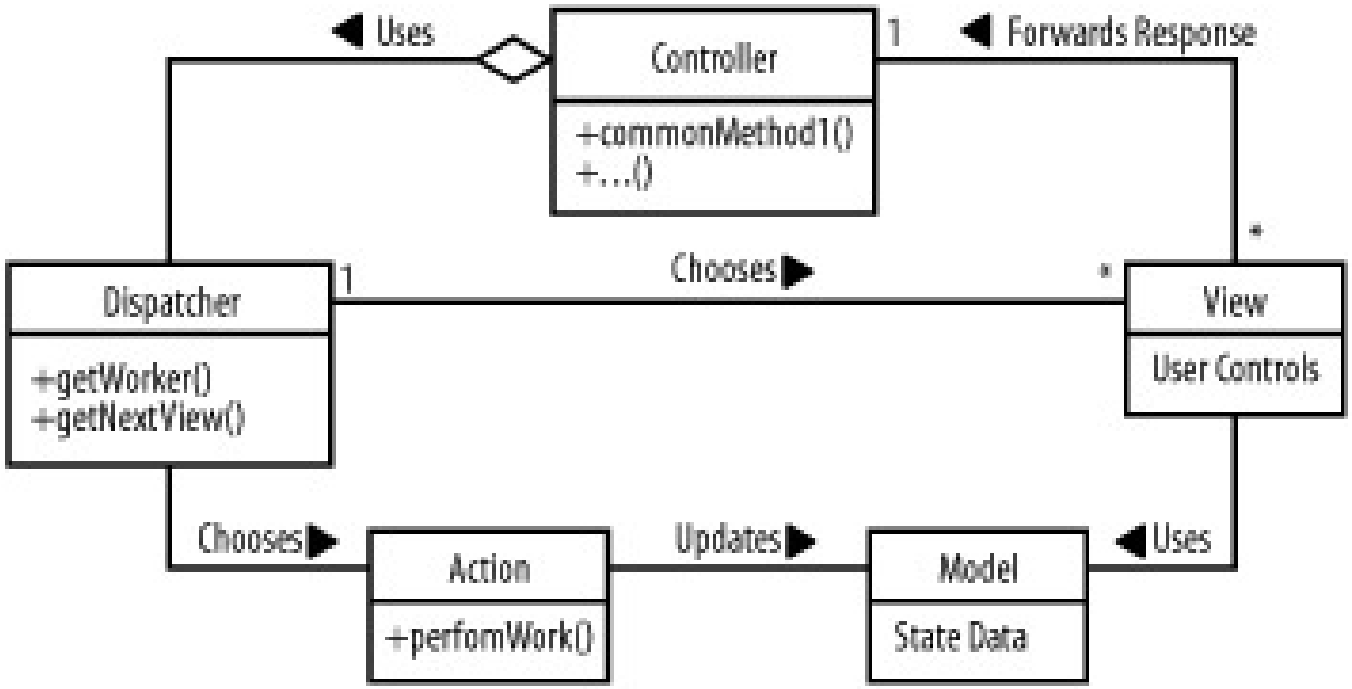
2 El patrón Modelo-Vista-Controlador

3 Diseño avanzado de la capa de presentación

El patrón *Service to Worker*

- Está basado en los patrones *Model-View-Controller* y *Front Controller*.
- El principal objetivo es mantener la separación entre las acciones, las vistas y los controladores.
- El servicio es el *Front Controller*, que se comporta como un punto único para gestionar las peticiones.
- El servicio delega la actualización del modelo a la acción asociada a una página, denominado el *worker*.
- En este patrón, el *dispatcher* es el objeto que lleva a cabo la tarea de manejar los *workers* y las vistas.
  - Encapsula la selección de páginas y los *workers*.
  - Desacopla el comportamiento de la aplicación del *Front Controller*.
  - Para modificar el orden en el que se muestran las páginas, sólo es necesario modificar el *dispatcher*.
- Este patrón es similar al *Front Controller*, pero con la adición de un *dispatcher*.

El patrón *Service to Worker*



Aplicación del patrón en J2EE

- Como ejemplo, imaginemos un *workflow*.
- Un *workflow* es una secuencia de tareas que se deben realizar en un determinado orden.
- Además queremos que la implementación sea extensible (añadir páginas y acciones sin modificar el *Front Controller*).
- Espeficiaremos el *workflow* en XML:

```
1 <?xml version="1.0" encoding="UTF−8"?>
2 <workflow>
3   <state name="login" action="LoginAction" viewURI="login.jsp" />
4   <state name="language" action="LanguageAction"
5                                          viewURI="language.jsp" />
6   <state name="display" action="RestartAction" viewURI="display.jsp" />
7 </workflow>
```

Simple *workflow*

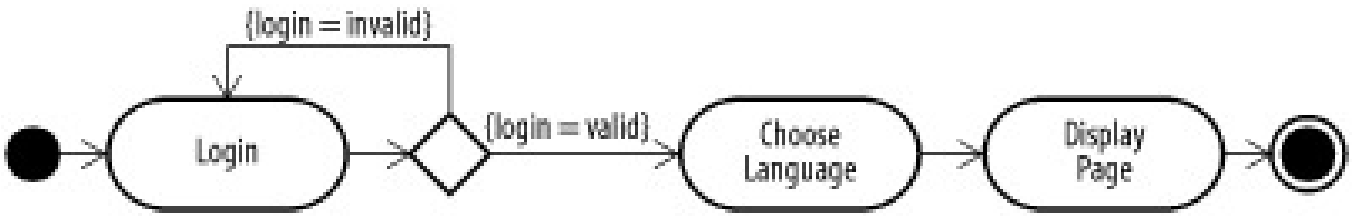


Diagrama de secuencia *workflow*