PROJECTE FI DE CARRERA

DISSENY I IMPLEMENTACIÓ D'UN MARC DE TREBALL DE PRESENTACIÓ PER A APLICACIONS J2EE

JAUME MARTÍ MUIXÍ

ENGINYERIA EN INFORMÀTICA

CONSULTOR:

JOSEP MARIA CAMPS RIBA

16/01/2012

LLICÈNCIA D'ÚS

A la meva dona Francesca que ha patit durant molts anys la meva obstinació per acabar aquesta carrera. Per tots els moments d'esbarjo que li he robat.

RESUM

En aquest projecte, s'estudia, analitza i implementa un framework per a la capa de presentació d'aplicacions J2EE, utilitzant el patró de disseny Model Vista Controlador (MVC) que es el recomanat per la presentació en aplicacions interactives. Aquest framework està pensat per ajudar als desenvolupadors d'aplicacions web amb J2EE a organitzar la presentació, a accedir a les funcions de la capa de negoci i a simplificar el flux de pàgines, el processament dels formularis i el control dels errors.

L'arquitectura Model Vista Controlador divideix la funcionalitat dels objectes involucrats en la presentació de dades per reduir al mínim el grau d'acoblament entre ells. Aquesta divisió es fa en tres capes: model, vista i controlador i separa clarament les seves respectives responsabilitats, cada capa s'encarrega tasques específiques.

El projecte està dividit en quatre fases:

- En la primera estudiarem aquest patró de disseny i altres patrons que ens seran útils en aquesta implementació, per esbrinar-ne el seu funcionament. També estudiarem les eines que ens proporciona J2EE com son els servlets i els JSP.
- En la segona analitzarem tres Frameworks que estan actualment al mercat i que utilitzen aquest patró per trobar els seus punt forts i febles i triar la millor manera d'implementar el nostre.
- En la tercera passarem a analitzar i implementar el nostre Framework.
- En la quarta dissenyarem i implementarem una petita aplicació que utilitzi el nostre Framework per provar el seu bon funcionament i la seva usabilitat.

El llenguatge de programació utilitzat serà el Java, l'entorn de desenvolupament serà l'eclipse i el servidor web serà l'apache tomcat. Totes aquestes tres tecnologies son estàndard i open source amb l'avantatge que suposa per a la seva distribució.

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

ÍNDEX

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

1 - INTRODUCCIÓ

1.1. DESCRIPCIÓ DEL PFC

En els últims anys s'està parlant dels sistemes distribuïts com del futur de l'enginyeria de programari. Dins dels sistemes distribuïts, les aplicacions web en son el principal representant tant per Internet com per a intranets, ja que, a més dels avantatges d'economia, fiabilitat i escalabilitat, aprofiten la potència i versatilitat dels navegadors i servidors web que hi ha al mercat. Aquests venen instal·lats en els ordinadors de sèrie, o son molt fàcils i econòmics d'aconseguir.

A mida que les aplicacions web s'han anat fent més grans i complexes, s'han anat emprant patrons d'anàlisi i de disseny cada cop mes estructurats i clars que han anat convertint una programació complexa i moltes vegades caòtica en una de clara i fàcil de mantenir. Un patró es una bona solució a un problema complex que es interessant aprofitar. Els avantatges de la utilització de patrons son:

- Evitar buscar reiteradament una solució al mateix problema
- Estandarditzar el disseny
- Facilitar l'aprenentatge

Hi ha patrons d'arquitectura, d'anàlisi i de disseny. Entre els patrons més importants hi ha els següents que ens interessa mencionar:

- El patró d'arquitectura en tres capes, que separa la capa de presentació, la capa de lògica de negoci i la capa d'accés a les dades. Utilitzant aquest patró només hem de tocar una de les capes si volem canviar de base de dades o volem canviar el disseny de les pantalles.
- El patró de disseny Model Vista Controlador (MVC), que implementa la capa de presentació. Aquest model es basa en separar la capa de presentació en tres mòduls:
    - El model representa la lògica de negoci i la base de dades i ve representat per una interfície on hi ha els mètodes i paràmetres que utilitzarem per accedir-hi.
    - La vista son les interfícies gràfiques amb les que interactua l'usuari.
    - El controlador es l'encarregat de regular el tràfic de l'usuari cap a la lògica de negoci i el retorn a la nova vista que s'ha de mostrar.

Mitjançant la utilització d'aquest patró s'aconsegueix incrementar la encapsulació permetent que la modificació d'un component no afecti als altres. A més, permet una major especialització dels perfils de desenvolupadors fent que cadascuna de les persones es dediqui a l'àrea més adequada a les seves competències.

- El patró Front Controller que centralitza les peticions d'entrada dels usuaris en un sol punt d'accés que gestionarà els permisos i delegarà les tasques als processos adequats. Aquest patró evita haver de definir en cada interfície d'usuari els accessos a la lògica de negoci i encapsula més la capa de presentació i la capa de lògica de negoci.
- View Helper que delega al controlador la construcció dels objectes i etiquetes que es veuen en les vistes. La utilització dels JSP es el principal exponent d'aquest patró.
- Dispatcher View que consisteix en delegar a un Dispatcher el flux de navegació, sent ell l'encarregat de decidir quina es la següent vista a mostrar.

Per ajudar a desenvolupar programari web que utilitzi aquest patró de disseny (MVC), han aparegut al mercat alguns Frameworks que proporcionen al desenvolupador una implementació del mateix. Un Framework o entorn de treball, es un conjunt de mòduls de programari que ajuden a un desenvolupador a unir els diferents components del projecte, utilitzant l'esquema conceptual del patró que implementa.

La utilització de Frameworks permet reduir el temps de desenvolupament d'un programari, la reutilització de components i la construcció dels mateixos seguint una política de disseny uniforme i utilitzant bones maneres.

En aquest projecte construirem un Framework que ajudi a la implementació d'aplicacions web desenvolupades en l'entorn J2EE bassades en el patró MVC.

Abans de començar la construcció, avaluarem tres propostes de Frameworks existents al mercat: Struts, Spring i JSF, veient-ne els diferents enfocaments i quines característiques tenen en comú. D'aquesta avaluació n' extraurem les característiques que voldrem que tingui el nostre Framework.

Aquests tres Frameworks son els més coneguts i utilitzats actualment i tenen suficients diferències perquè el seu estudi ens aporti els coneixements necessaris.

Struts va ser construït per Apache Software Foundation dins el projecte Jakarta, però ara es un projecte independent conegut com Apache Struts.

Spring va ser construït inicialment per Rod Johnson, encara que en aquest moment hi ha més de 400 desenvolupadors continuant el projecte.

Java Server Faces (JSF) va ser construït per Sun Microsystems que encara l'està mantenint en l'actualitat.

Després d'analitzar i construir els components de l'entorn de treball, crearem una petita aplicació per mostrar com s'utilitza.

1.2 OBJECTIUS DEL PFC

L'objectiu principal del projecte es construir un entorn de treball per a la capa de presentació d'aplicacions J2EE que implementi el patró de disseny Model Vista Controlador (MVC).

Aquest Framework ha de complir els següents requisits:

- S'ha de construir utilitzant patrons de disseny i seguint les bones pràctiques recomanades en la construcció de programari orientat a objectes.
- Ha d'incorporar les millors característiques (o les que més ens agraden) dels tres Frameworks més coneguts que hi ha actualment al mercat.
- Ha de ser robust, fàcil d'usar per part dels programadors i ha de permetre fer modificacions sobre la part ja construïda també amb facilitat.

Un objectiu addicional fora de l'abast d'aquesta part del projecte, però que s'ha de realitzar en un curt termini es la construcció d'un Plugin per poder utilitzar aquest Framework en el IDE Eclipse.

1.3 PLANIFICACIÓ DEL PROJECTE

Aquest projecte es composa de les següents fases:

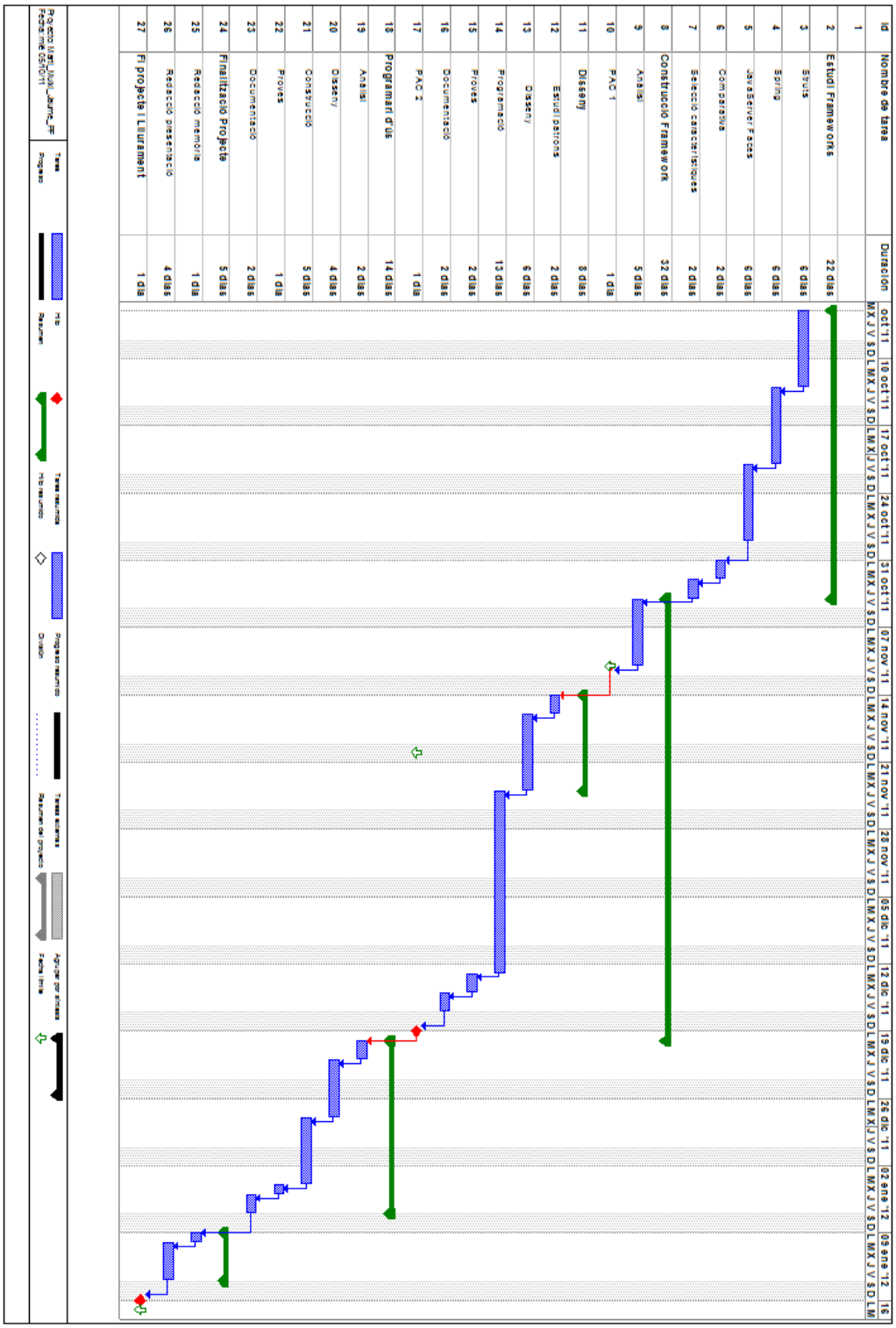
1. Estudi de Frameworks del mercat
   1.1. Struts
   1.2. Spring
   1.3. JavaServer Faces
   1.4. Comparativa
   1.5. Selecció de les característiques del nostre Framework
2. Construcció del Framework
   2.1. Anàlisi
   2.2. Disseny
      2.2.1. Estudi dels patrons a utilitzar
      2.2.2. Disseny
   2.3. Programació
   2.4. Proves
   2.5. Documentació
3. Construcció d'un programari utilitzant el Framework
   3.1. Anàlisi
   3.2. Disseny
   3.3. Construcció
   3.4. Documentació
   3.5. Proves del programari resultant
4. Finalització del projecte
   4.1. Redacció de la memòria
   4.2. Redacció de la presentació

Hi ha tres fites o lliurament:

- PAC 2 – 10 Nov. – Ha de coincidir amb la fi de l'anàlisi (punt 2.1) i s'ha de lliurar un document Que contindrà l'estudi dels tres Frameworks, la seva comparativa, l'enumeració de les característiques seleccionades i l'anàlisi de com el construirem.
- PAC3 – 19 Des. – Ha de coincidir amb la fi de la construcció del Framework (punt 2) i s'ha de lliurar l'estudi dels patrons, els diagrames de disseny, el codi font, les proves realitzades i la documentació.
- Final – 16 Gen. – Ha de coincidir amb la finalització del projecte (punt 4) i s'ha de lliurar, l'anàlisi i els diagrames de disseny del programari de prova, la documentació i el codi font resultant i el resultat de les proves al programari construït. També es lliurarà la memòria i la presentació.

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

Diagrama de Gant



Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

1.4 PRODUCTES OBTINGUTS

D'aquest projecte se n'obtindrà cinc productes o Lliurables:

1 Codi font i executable del programari del Framework
Conjunt de classes que formen el codi font del programari i conjunt d'empaquetat d'objectes que formen l'executable del programari.
2 Codi font del programari per utilitzar el Framework
Conjunt de classes que formen el codi font obtingut al utilitzar el Framework.
3 Documentació del programari
On s'explicarà els requisits per al bon funcionament del Framework, la seva instal·lació i com s'utilitza explicant com s'ha construït el programari d'exemple.
4 Memòria del projecte
Explicació exhaustiva del projecte
5 Power Point de Presentació del projecte.
Explicació gràfica del projecte ressaltant-ne les parts més importants.

1.5 BREU DESCRIPCIÓ DELS ALTRES CAPÍTOLS.

La resta de capítols seguiran una estructura similar a la que hem utilitzat a l'hora de fer la planificació per descriure les fases del projecte:

2 Estudi de Frameworks del mercat
- 2.2 Struts
- 2.3 Spring
- 2.4 JavaServer Faces
- 2.5 Comparativa
- 2.6 Selecció de les característiques del nostre Framework

3 Construcció del Framework
- 3.2 Anàlisi
- 3.3 Disseny
  - 3.3.1 Estudi dels patrons a utilitzar
  - 3.3.2 Disseny
- 3.4 Programació
- 3.5 Proves

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

2 - ESTUDI DE FRAMEWORKS DEL MERCAT

2.1 INTRODUCCIÓ

El ***patró Model-Vista-Controlador (MVC)*** separa la lògica de negoci, la presentació, i les accions basades en la entrada de l'usuari en tres classes separades:

Model . El model maneja el comportament i les dades del domini d'aplicació, respon a les sol·licituds d'informació sobre el seu estat (en general de la vista), i respon a les instruccions per canviar l'estat (en general des del controlador).

Vista . Visualització de la informació.

Controlador . El controlador interpreta les entrades de l'usuari i la informació del model a fi de canviar el necessari.

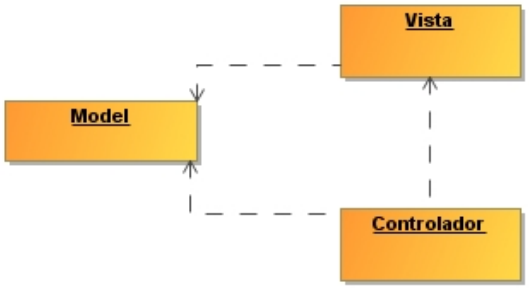


Figura 1: Estructura MVC

És important assenyalar que tant la vista com el controlador depenen del model, però el model no depèn ni de la vista ni del controlador. Aquest és un dels principals avantatges de la separació. Aquesta separació permet que el model es construeixi i es provi independentment de la presentació visual. La separació entre la vista i el controlador és difícil en molts casos, i de fet, molts Frameworks no web implementen les dues funcions en un sol objecte. En les aplicacions web, la separació entre la vista (el navegador) i el controlador (els components del costat del servidor que controla la petició HTTP) està molt ben definida.

Ens els propers apartats d'aquest capítol veurem tres maneres diferents d'abastar aquest problema.

2.2 STRUTS I STRUTS2

2.2.1. STRUTS

Struts forma part del projecte Jakarta de la Apache Software Foundation. Inicialment el van començar a desenvolupar 30 programadors els anys 2000 i 2001 seguint les directrius de Craig. R. MacClanahan, que també va dissenyar el Tomcat i el paquet de Web Services per a Java. Actualment, Craig està encapçalant els projectes JSF i J2EE a Sun Microsystems. Struts es distribueix com a software Obert.

Struts es un controlador MVC2 que té la següent arquitectura:

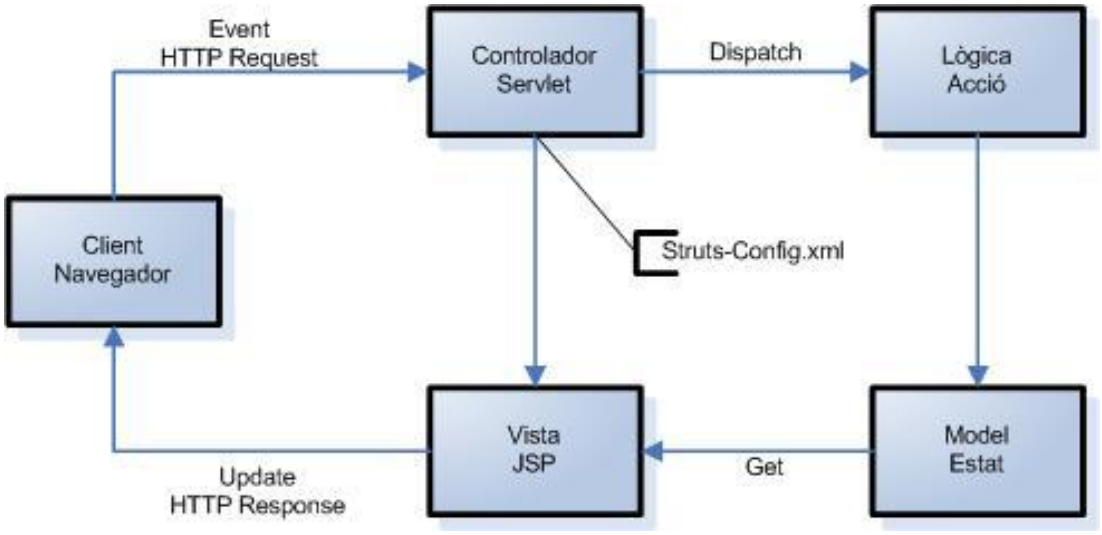


Figura 2: Funcionament General Struts

- **Navegador del client**
  Una petició HTTP des del client (HttpRequest) crea un esdeveniment. El contenidor Web li respondrà amb una resposta HTTP (HttpResponse).

- **Controlador (ActionServlet)**
  El controlador rep la petició des del navegador i pren la decisió d'on enviar la sol·licitud.

Struts implementa el controlador com un Servlet, per tant, utilitza el patró de disseny FrontController (totes les peticions es reben en un sol punt que s'encarrega de distribuir-les). També utilitza el patró de disseny comandament (utilitza un objecte per encapsular diferents accions), això permet superar elegantment la ambigüitat de la comanda executar una acció.

Per saber quina acció triar, el controlador (Servlet) utilitza el fitxer Struts-config.xml. La informació d'aquest fitxer es carrega al arrancar el programa en un conjunt d'ActionMapping que es posen en un contenidor ActionMapper. Un ActionMapping porta el nom complert d'una classe d'una acció perquè el patró comandament la carregui, i lliga les possibles respostes de la acció amb la vista (JSP) corresponent.

- **La lògica**
  La lògica del negoci actualitza l'estat del model i ajuda a controlar el flux de l'aplicació. Amb Struts això es fa amb una classe d'acció, totes les accions son una subclasse de la classe Action. Cada acció encapsula les crides a la classe de lògica de negoci real, interpreta els resultats i dóna la resposta perquè s'enviï el control al component de vista apropiat.

- **L'estat del model**
  Les accions actualitzen l'estat de l'aplicació i obtenen una classe **ActionForm** que representa l'estat del model en una sessió o nivell de sol·licitud. Cada acció té el seu Form assignat i els JSP llegeixen la informació de la classe ActionForm mitjançant etiquetes.

- **Vista**
  La vista és un arxiu JSP que utilitza etiquetes JSP i etiquetes especials creades per Struts per formar la pàgina resposta agafant les dades des del ActionForm. Les etiquetes pròpies són una de les coses que fan Struts únic en comparació amb altres Frameworks, amb aquestes etiquetes els responsables de la presentació han d'utilitzar molt poc codi Java. Struts va separar clarament els JSP de la capa de negoci i els JSP extreuen la informació directament dels formularis (FORMS).

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

2.2.2. STRUTS2

Per reduir la complexitat a l'hora de desenvolupar aplicacions en Struts va néixer **Struts2**. Struts2 es va construir unint esforços entre els desenvolupadors d'Struts i els d'un nou Framework, el WebWork2. Struts 2 ofereix una millora en la productivitat dels desenvolupadors que l'utilitzen degut a les següents característiques:

- Utilitza menys XML canviant-lo per anotacions Java.
- Defineix comportaments per defecte.
- Permet utilitzar qualsevol classe com una acció.
- Té una estructura molt modular amb un grau d'acoblament molt baix.
- Converteix automàticament i de forma transparent utilitzant getters i setters el mapeig entre valors HTTP i dades de les accions.
- Els fitxers de configuració son modulars i es poden des composar en diversos fitxers per facilitar-ne el manteniment.
- Utilitza temes, plantilles i etiquetes reutilitzables.
- Amplia molt la biblioteca d'etiquetes pròpies.

Per a rebre les sol·licituds del navegador, no s'utilitza el servlet ActionServlet sinó que s'utilitza el **FilterDispatcher** que també implementa el patró FrontControler i a més implementa un filtre estàndard J2EE. També utilitza el ActionMapper per saber si s'ha de cridar alguna acció i en cas afirmatiu, a quina acció ha de cridar.

Les sol·licituds en Struts2 son completament diferents de les de Struts i segueixen la lògica següent:

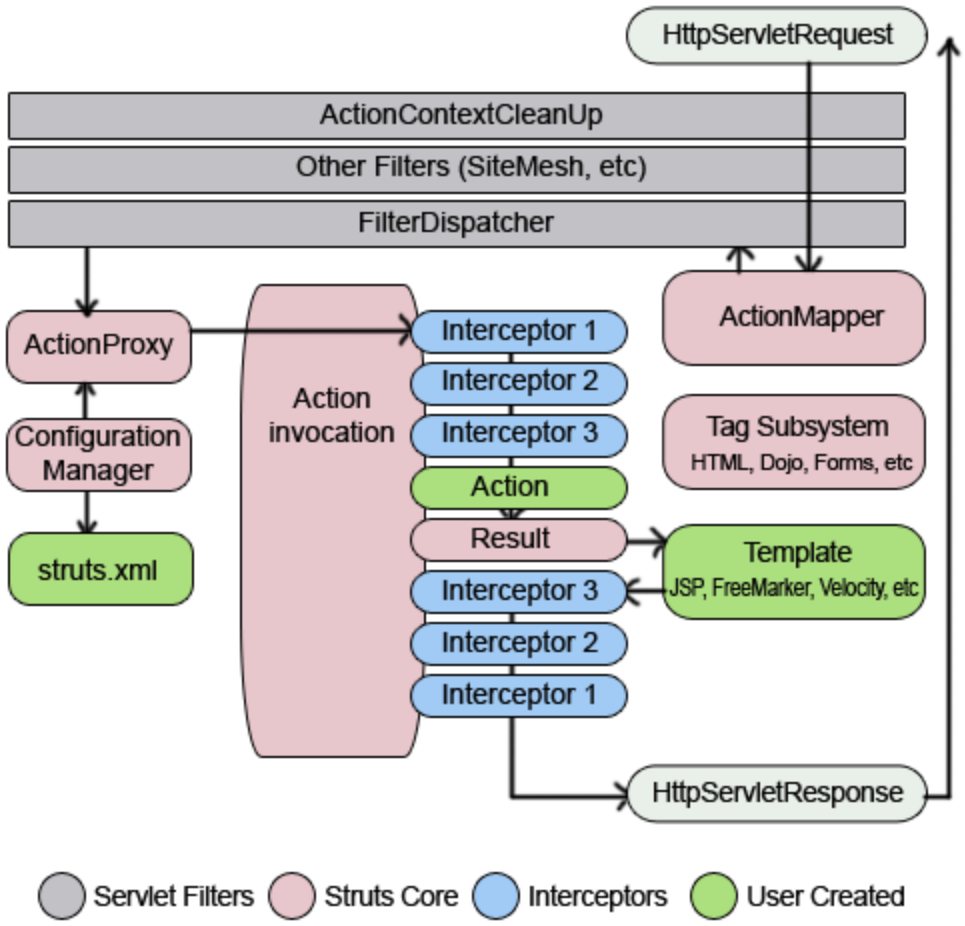


Figura 3: Processament d'una sol·licitud amb Struts2

ActioncontextCleanUp i Other Filters s'utilitzen només per poder integrar Struts amb altres tecnologies.

Si la sol·licitud d'un client coincideix amb el nom d'alguna acció, **FilterDispatcher** delega aquesta tasca a **ActionProxy**, que es basa en els arxius de configuració començant pel struts.xml. **ActionProxy** crea un **ActionInvocation** que implementa el patró comandament per crear la acció corresponent.

ActionInvocation invoca els **interceptors** si està així configurat i després invoca la **acció**. A continuació el **resultat** s'executa presentant JSP o plantilles.

Després els **interceptors** s'executen en mode invers i finalment la resposta torna a través dels filtres configurats al arxiu web.xml.

Els **interceptors** es poden configurar per cada **acció** i proporcionen diferents capes de funcionalitat, per exemple control de flux o seguretat o validació de dades.

Una **acció** a Struts2 no estén cap altre classe o interfície, pot implementar la interfície Action que té el mètode Execute(), però no es obligatori. La acció ha de tenir un mètode Execute(), en cas contrari la invocació a la acció dóna un error. El mètode Execute() no rep cap paràmetre i retorna un String, llavors, perquè la acció rebi els paràmetres, s'utilitza la tècnica d'injecció de dependències implementada com una injecció de setter, així, per obtenir objectes només cal crear un mètode setter per cada un. Per als objectes que no son de negoci (per exemple ServletRequest) hi ha una interfície definida que té per nom el nom de l'objecte + aware (per exemple public interface ServletRequestAware) que conté el mètode setter de l'objecte. Per obtenir aquest objecte s'ha d'implementar la Interfície.

Els **interceptors** son una funcionalitat molt potent a Struts2 i permeten executat codi abans i després de la acció o evitar que la acció s'executi. Es solen utilitzar per implementar funcionalitat comú a moltes accions que s'executa fora de l'acció. Els interceptors son classes que segueixen el patró Interceptor:

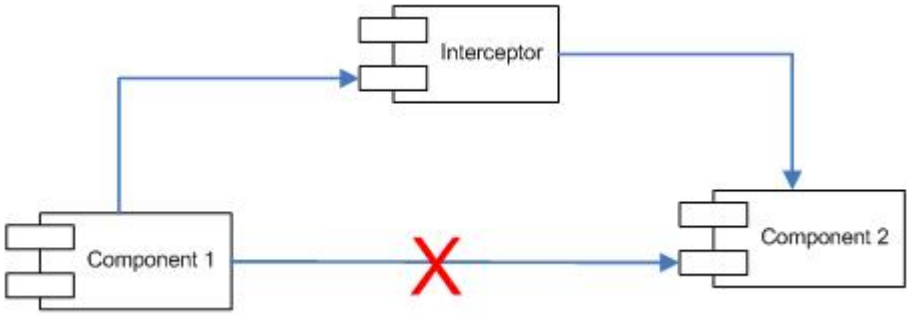


Figura 4: Funcionament dels interceptors

Que el que vol oferir es una manera de canviar o incrementar el procés estàndard.

Struts2 permet crear piles d'interceptors i aplicar-les a totes les accions en l'ordre que apareixen a la pila.

Result es un mecanisme perquè les accions puguin retornar diferents tipus de resultats i no només JSP, així, es poden integrar diferents eines de presentació. Existeixen pluggins que proporcionen diferents tipus de resultats, el tipus de resultat per defecte es dispatcher. Una acció pot tenir més d'un result associat i triar quin s'envia depenent del resultat de la execució de la acció.

Per fer arribar els paràmetres des de la acció fins la vista s'utilitza una pila (ValueStack). Les vistes tenen sempre accés a la pila

2.3 SPRING

Spring va ser inicialment llençat el juny del 2003 sota llicència Apache i la primera versió va ser distribuïda el març del 2004.

Spring ofereix una gran llibertat als desenvolupadors de Java i a més té solucions àgils i ben documentades per pràctiques habituals a la industria.

Es un Framework molt gran, per això s'ha dividit el projecte en 20 mòduls:

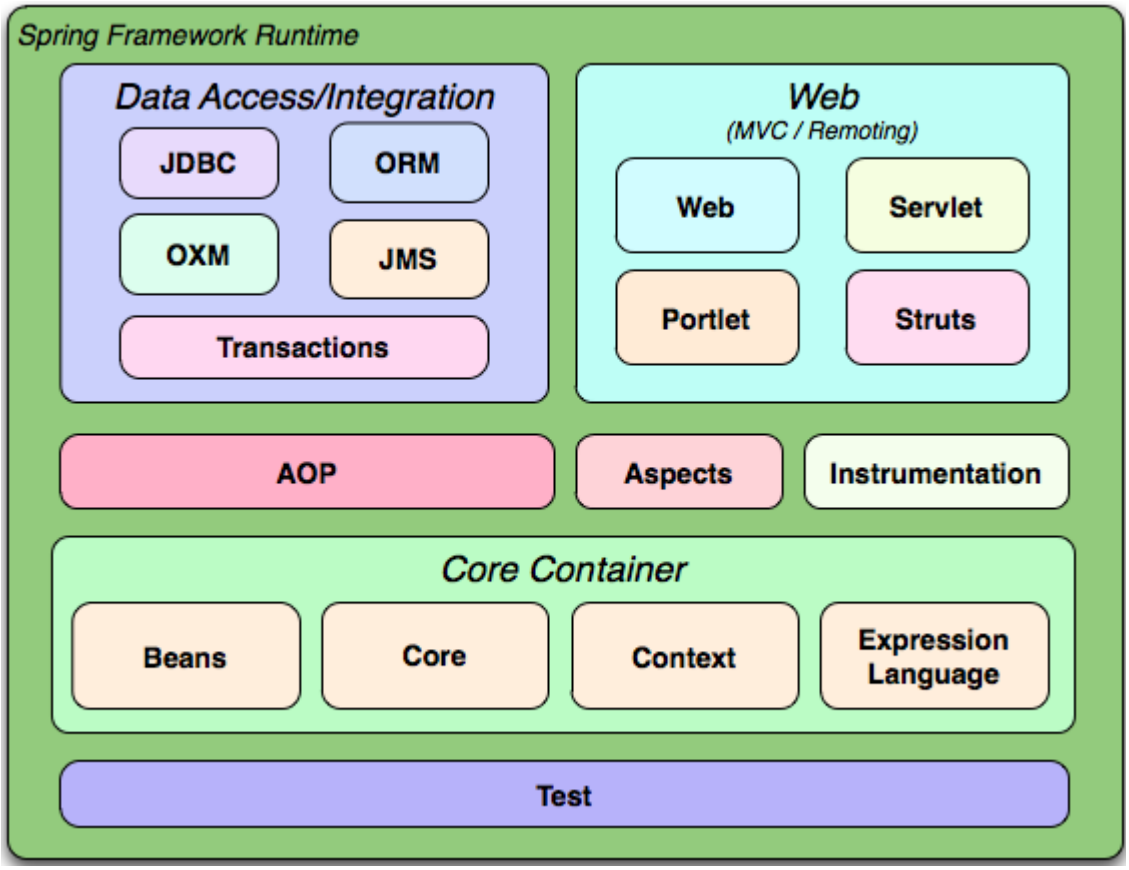


Figura5: Mòduls de que es composa Spring

Aquests mòduls no cal utilitzar-los tots, es pot utilitzar Spring amb Struts i no utilitzar el mòdul MVC o es pot utilitzar Spring amb Hibernate que es connectaria a Spring mitjançant el mòdul ORM.

Aquests mòduls es poden agrupar en nou mòduls importants:

Contenidor d'Inversió de Control :

Es el mòdul principal d'Spring que utilitza inversió de control i injecció de dependències per la configuració i gestió d'objectes Java. S'encarrega de crear i configurar objectes, cridar mètodes d'inicialització o passar objectes per crides registrades a altres objectes. Aquesta és una de les aportacions més importants que proveeix. Les definicions dels objectes a crear estan en fitxers XML i els objectes els passa utilitzant mètodes constructors, propietats o mètodes factoria.

En molts casos no cal usar el contenidor quan usem altres parts d'Spring Framework encara que usant-lo probablement farem que l'aplicació sigui més fàcil de configurar i personalitzar.

Entorn de Programació orientada a aspectes (AOP):

Spring té el seu propi entorn de programació orientada a l'aspecte que modularitza problemàtiques multicapa en aspectes. L'entorn AOP està basat en la intercepció i es configura en temps d'execució, això elimina la necessitat de certs passos de compilació o de temps de càrrega de l'aplicació.

Entorn d'accés a les dades:

L'entorn d'accés a les dades d'Spring, s'adreça a problemàtiques comuns a les que els desenvolupadors s'enfronten quan treballen amb les bases de dades. Dóna suport a JDBC, Hibernate, iBatis, JDO, JPA, Toplink, Apache OJB i Cayenne (Java) entre altres. Per tots aquests entorns, Spring aporta les funcionalitats següents: Gestió de recursos, Gestió d'excepcions, Participació de la transacció, Unwrapping de recursos, Abstracció per la gestió de BLOB i CLOB.

Totes aquestes funcionalitats esdevenen disponibles quan usem classes Template proveïdes per Spring per cada entorn suportat.

Entorn de gestió de transaccions:

Aporta un mecanisme de transacció de la plataforma Java. Es capaç de treballar amb transaccions locals i globals i treballar amb transaccions niuades. S'utilitza per transaccions amb connexió JDBC, transaccions sobre Unitats de Treball de Mapeig d'Objecte Relacional, transaccions sobre JTA o transaccions sobre altres recursos com ara, bases de dades orientades a objectes.

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

Entorn d'accés remot:

Es una abstracció per treballar amb diferents tecnologies basades en RPC disponibles en Java, tant per connectivitat del client com per exportar objectes a servidors. Aporta recuperació a fallades (reconnexió automàtica) i algunes optimitzacions per ús a la part de client de sessions Beans sense estat remots EJB. Suporta protocols basats en HTTP, Hessian, RMI, RMI-IIOP (Corba), Enterprise javaBean. EJB, SOAP, serveis web Apache Axis

Entorn d'autenticació i personalització:

Autenticació i processos d'autorització que suporten molts estàndards de la indústria, protocols, eines i pràctiques via el subprojecte Acegi security framework.

Entorn de missatgeria:

Millora d'enviament de missatges sobre APIs JMS estàndards.

Entorn de testeig:

Suporta classes per la creació d'unitats de testeig i proves d'integració.

Entorn model-vista-controlador:

Es la part que ens interessa i la que realment volem estudiar d'aquest Framework.

Es un entorn basat en peticions comparable a Struts. Està basat en un DispatcherServlet que s'encarrega de resoldre les peticions.

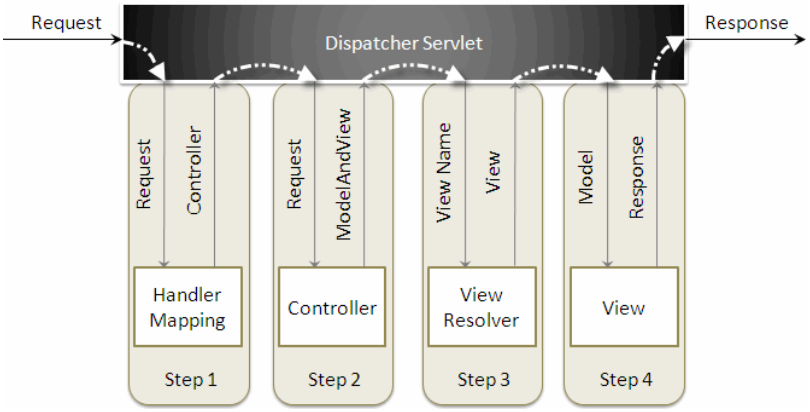


Figura6: Flux d'una petició al DispatcherServlet

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

Quan s'envia una sol·licitud a Spring esdevé la seqüència d'esdeveniments que es mostren a la figura 6:

- El DispatcherServlet rep la sol·licitud
- El DispatcherServlet consulta al HandlerMapping i invoca el controlador associat a la sol·licitud.
- El controlador processa la petició cridant els mètodes apropiats de la lògica i retorna un ModelAndView al DispatcherServlet. L'objecte ModelAndView conté les dades del model i el nom de la vista.
- El DispatcherServlet envia el nom de la vista al ViewResolver per trobar la vista a invocar.
- El DispatcherServlet passa les dades del model a la vista per obtenir la vista resultat.
- La vista amb les dades del model torna el resultat a l'usuari.

Hi pot haver més d'un DispatcherServlet en una aplicació. Al fitxer web.xml es defineixen els possibles DispatcherServlet de la següent manera:

```
<servlet>
        <servlet-name>vendes</servlet-name>
        <servlet-
        class>org.springframework.web.servlet.DispatcherServlet</servletclass>
        <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
        <servlet-name>vendes</servlet-name>
        <url-pattern>*.vta</url-pattern>
</servlet-mapping>
```

Així totes les peticions que acaben amb .vta utilitzen el DispatcherServlet vendes. Al iniciar cada DispatxerServlet busca en el directori WEB-INF el fitxer nomServlet-servlet.xml.

Els DespatcherServlet utilitzen un WebApplicationContext per configurar-se. Cada DispatcherServlet té el seu WebApplicationContext que hereta d'un WebApplicationContext genèric.

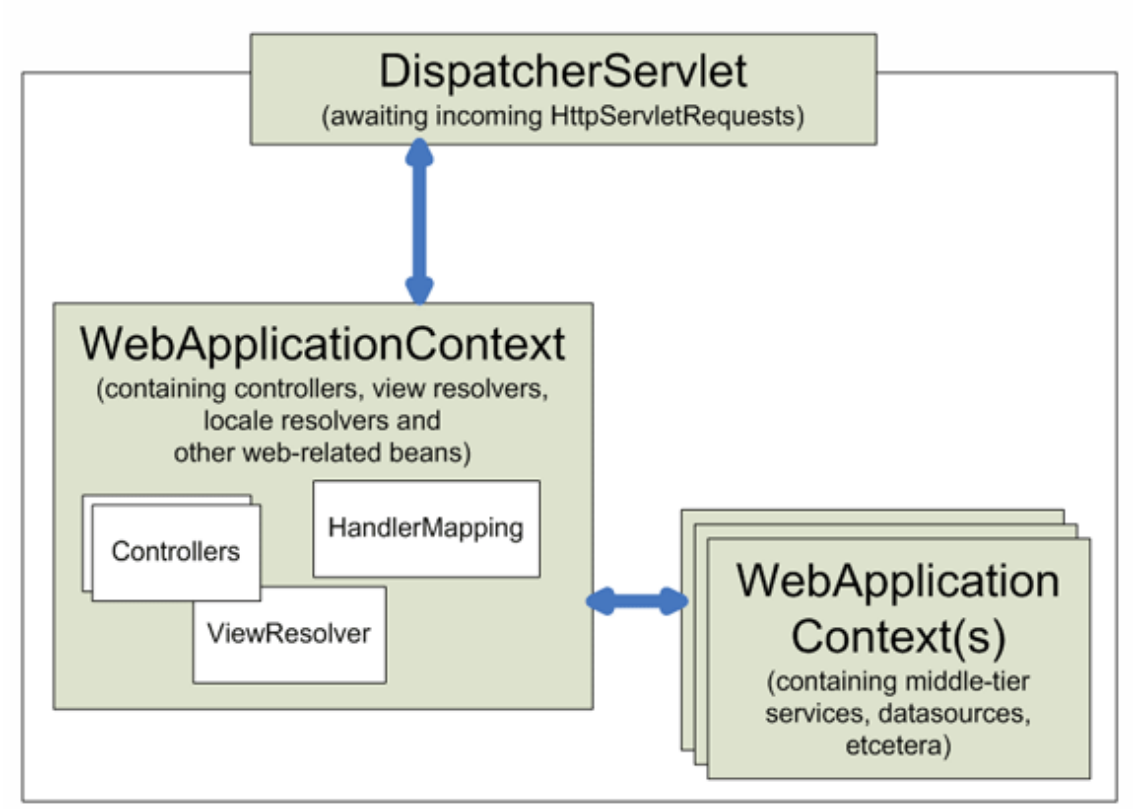


Figura7: Configuració de cada DispatcherServlet

D'aquest WebApplication Context proporciona els HandlerMapping, Els Controllers i els ViewResover que necessita.

HandlerMapping

El HandlerMapping o assignació de controlador lliura un HandlerExecutionChain que conté el controlador que coincideix amb la petició d'entrada i la llista d'interceptors, si n'hi ha, que s'apliquen a la sol·licitud. Si conté interceptors, aquests es podran executar abans que el controlador, després o en ambdues ocasions.

Des de la versió 2.5 Spring ofereix el DefaultAnnotationHandlerMapping que no cal implementar-lo i busca anotacions @RequestMapping dins les anotacions @controllers.

Controllers

Els controladors accedeixen a la lògica de la aplicació real definit en una interfície de servei. Spring implementa un controlador d'una manera molt abstracta, de manera que permet crear una amplia varietat de controladors.

En la versió 2.5 Spring va introduir el concepte de programació basada en anotacions en el Servlet. Les anotacions son del tipus @RequestParam, @RequestMapping, @ModelAttribute. Els controladors implementats en aquest estil no han d'estendre cap classe ni implementar cap interfície.

```
@Controller
públic  de classe HelloWorldController {
        @RequestMapping ( "/HelloWorld" )
        públic ModelAndView helloWorld () {
    ModelAndView mav = new ModelAndView ();
    mav.setViewName ( "helloWorld" );
    mav.addObject ( "message" , "Hello World!" );
    return mav;
        }
}
```

Com es pot veure, les anotacions @Controler i @RequestMapping permeten noms de mètodes i signatures flexibles. En el cas anterior, el controlador no rep paràmetres i torna un ModelAndView, però no ha de ser sempre així. ModelAndView, @Controller i @RequestMapping son la base de les implementacions de Spring. La anotació @RequestMapping es pot ficar a nivell de classe així:

```
@Controller
@RequestMapping ( "/HelloWorld" )
public class HelloWorldController {
        @autowired
        public HelloWorldControler (){}

        @RequestMapping (method = RequestMethod.GET )
        public ModelAndView helloWorld () {
    ModelAndView mav = new ModelAndView ();
    mav.setViewName ( "helloWorld" );
    mav.addObject ( "message" , "Hello World!" );
    return mav;
        }
```


Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

```
        @RequestMapping (value = "/{day}", method = RequestMethod.GET )
        public ModelAndView helloWorld (@PathVariable @DateTimeFormat(iso) Date day) {
    ModelAndView mav = new ModelAndView ();
    mav.setViewName ( "helloWorld" );
    mav.addObject ( "message" , "Hello World! Today "+day );
    return mav;
        }
}
```

En aquest cas si s'accedeix a /HelloWord pel mètode POST sense paràmetres s'executa el primer mètode i si s'hi accedeix amb el paràmetre data s'executa el segon. Els HandlerMethod poden retornar els següents tipus: ModelAndView, Model, Map, View, String (nom de la vista), HttpEntity o ResponseEntity.

ViewResolver

Spring permet utilitzar com a vista JSP, velocicy i XSLT. Per tractar les vistes Spring proporciona dos interfícies importants: ViewResolver i View. El ViewResolver proporciona la View a partir del seu nom i la View prepara la resposta per a la tecnologia que s'utilitzi. Hi ha diferents implementacions de ViewResolver:

ViewResolver	Descripció
`XmlViewResolver`	Implementació de `ViewResolver` que accepta un fitxer de configuració XML. El fitxer de configuració per defecte es `/WEB-INF/views.xml`.
`UrlBasedViewResolver`	Simple implementació de `ViewResolver` que efectua una resolució directa de noms lògics de View a URLs. Es apropiada si els noms coincideixen amb la URL de la vista.
`InternalResourceViewResolver`	Subclasse de `UrlBasedViewResolver` que dóna suport a Servlets, JSP, `JstlView` i `TilesView`.
`VelocityViewResolver` / `FreeMarkerViewResolver`	Subclasse de `UrlBasedViewResolver` que dóna suport a `VelocityView`.
`ContentNegotiatingViewResolver`	Implementació de `ViewResolver` que resol una View basada en HttpEntity o ResponseEntity

Figura8: Taula d'implementacions de ViewResolver

2.4 JAVASERVER FACES (JSF)

Java Server Faces (JSF) es el Framework per a aplicacions Web en Java de Sun Microsystems alliberat el març del 2004 i vol ser un estàndard de presentació d'aplicacions web. JSF està orientat a la interfície gràfica d'usuari (GUI) per facilitant-ne el desenvolupament. La filosofia de JSP es similar a la de Swing per aplicacions Java estàndards, basant-se igualment en components.

Els components de la pàgina son representats com objectes amb estat propi, el que permet al servidor manipular l'estat i connectar esdeveniments. Aporta els components bàsics (botons, llistes, etc) i permet crear-ne de més complexos (menús, pestanyes, etc). Permet fer un arbre de components continguts els uns en els altres, per exemple un formulari conté una pestanya que conté un botó. Els components poden reaccionar a diferents esdeveniments com un canvi de valor d'un camp o passar per damunt d'una imatge

A més, separa la lògica de la presentació proporcionant el seu propi Servlet que fa de controlador, implementant el patró de disseny Model-View-Controller (MVC). Permet definir la lògica de navegació a partir d'unes regles emmagatzemades en arxius de configuració faces-config.xml. aquestes regles son del tipus: si venim des d'un esdeveniment d'una pàgina i obtenim un cert resultat llavors hem d'anar a tal altra pàgina.

El funcionament de JSF es molt senzill:

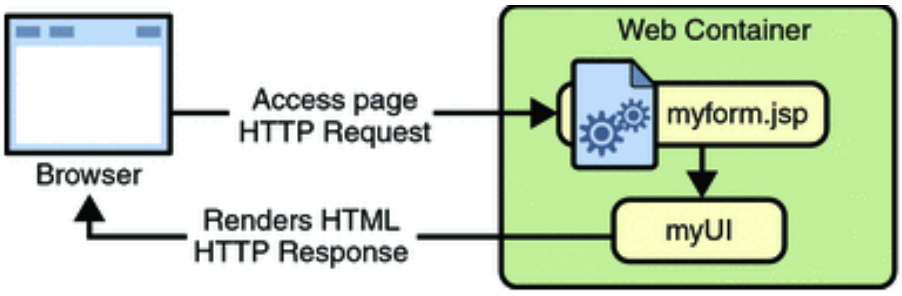


Figura9: Funcionament de JSF

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

myform.jsp es una pàgina jsp que conté etiquetes específiques de JSF. En aquesta pàgina hi ha els components d'interfície d'usuari definits per aquestes etiquetes específiques.

myUI s'encarrega de manegar els objectes referenciats en el formulari.

Java Server Faces proporciona es següents elements:

- **Un conjunt de UIComponents** i interfícies de comportament que especifiquen l'estat i la funcionalitat dels components de la interfície d'usuari. Entre aquest conjunt de components hi ha:
  - UIData que representa un enllaç de dades
  - UIColumn que representa una columna d'un enllaç de dades
  - UICommand que executa accions quan s'activa
  - UIForm Encapsula un grup de controls que envien dades al servidor.
  - UIGrafic que mostra una imatge
  - UIInput que s'utilitza perquè l'usuari introdueixi dades.
  - UIMessage que mostra un missatge
  - UIOutput que mostra dades en una pàgina
  - UIPanel que maneja el disseny dels components que incorpora.
  - UISelect Boolean que es una casella de selecció
  - UISelectItem que permet seleccionar un element d'un conjunt
  - UISelectMany que permet seleccionar diversos elements d'un conjunt
  - Etcetera.

- **Un model de representació** que defineix com utilitzar el mateix component de diferents maneres: Es pot canviar el disseny d'un component seleccionant diferents etiquetes, així, el mateix UIComponent UICommand es pot cridar amb l'etiqueta commandButton que representa un botó o amb l'etiqueta commandLink que representa un enllaç.

- **Un model d'escolta d'esdeveniments** que defineix com tractar els esdeveniments dels components: Quan l'usuari activa un component s'activa un esdeveniment, això fa que la implementació del JSF invoqui el mètode Listener que processi aquest esdeveniment.

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

JSF suporta tres tipus d'esdeveniment:

- Una acció, que es produeix quan s'activa un component.
- Una variació de valor, que es produeix quan l'usuari canvia el valor d'un component. Exemple canviar el valor d'un UIInput o canviar la selecció en qualsevol UISeect
- Un canvi de dada que es produeix quan es selecciona una nova fila en un UIData

- **Un model de conversió** que defineix com registrar els convertidors de dades en un component.

- **Un model de validació** que defineix com registrar validadors en els components. JSF permet validar dades locals abans de ser actualitzades, aquesta validació es fa utilitzant etiquetes JSF que implementen validadors estàndards. Per exemple l'etiqueta LongRangeValidator comprova si un valor es troba dins un rang i l'etiqueta LengthValidator comprova que la longitud d'un String estigui dins un rang.

La lògica de **navegació** entre pàgines de JSF es molt senzilla i molt fàcil definir, es basa en un conjunt de regles que es defineixen en els fitxers de configuració usant petits conjunts d'elements XML. Una regla de navegació es definiria:

```
<navigation-rule>
  <from-view-id>/greeting.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/response.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

En aquest cas quan a la pàgina greeting.jsp es desencadena l'esdeveniment success s'ha d'anar a la pagina response.jsp.

En la pàgina greeting.jsp el botó pot tenir diferents configuracions, així:

```
<h:commandButton id="submit" action="success" value="Submit" />
```

Provoca l'esdeveniment success sempre, en canvi:

```
<h:commandButton id="submit" action="#{userNumber.getStatus}" value="Submit" />
```

Quan l'usuari prem el botó es genera un esdeveniment, aquest esdeveniment es tractat per default actionListener, que crida el mètode descrit en la acció del component. Aquest mètode retorna al listener el resultat lògic, el qual passa aquest resultat més una referència al mètode al default navigatioHandler que selecciona la pàgina a mostrar.

Cicle de vida d'una pàgina JSF

El cicle de vida d'una pàgina JSF es similar a la d'una pàgina JSP, donat que, en ambdós casos el client fa una sol·licitud amb un HTTP request i el servidor li respon amb una pàgina en format HTML. No obstant, JSF es més complex i es necessiten sis fases per poder donar suport al model de components, a la validació de les dades i a la conversió.

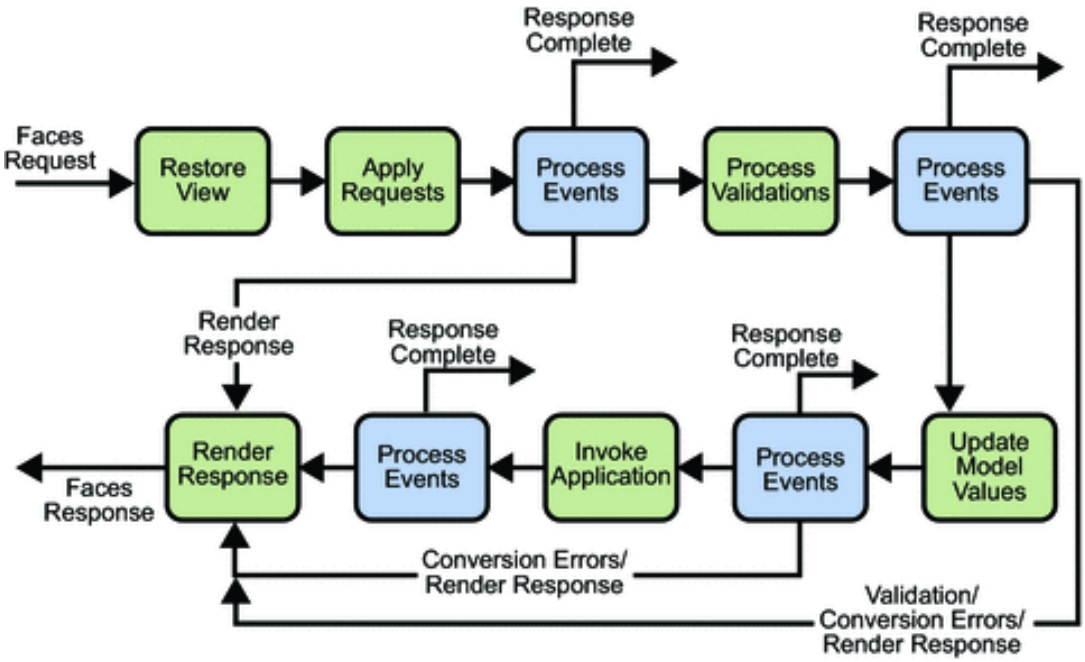


Figura10: cicle de vida d'una sol·licitud JSF

A la figura 9 es veu en color verd les sis fases del procés i en color blau el moment en que aquestes fases processen esdeveniments. A continuació s'explica que es fa en cada una de les fases:

1 – Restaurar vista: En aquesta fase es construeix un arbre de components i s'assignen els validadors i listeners declarats.

2 – Assignar valors: En aquesta fase s'assignen a partir dels paràmetres els valors a cada un dels components de l'arbre, realitzant les conversions necessàries. Si no hi ha cap esdeveniment a processar es salta a RenderResponse.

3 – Validacions: En aquesta fase es contrasten les dades amb les regles de validació que s'han introduït. Si algun valor es incorrecte, es genera un error i es salta a RenderResponse.

4 – Actualització dels valors: En aquesta fase es fa la conversió de tipus. Si alguna conversió falla es genera el missatge d'error corresponent i es passa al Render Response.

5 – Invocació de la lògica de negoci: En aquest pas es crida a la lògica de negoci i es calcula la nova vista a enviar a partir de les regles que hi ha al faces-config.xml i del resultat retornat.

6 – Fase d'enviament de la vista de resposta: En aquesta fase s'executa la vista (Un JSP normalment) i es construeix l'arbre de components que serà utilitzat en la fase de restaurar vista de a propera petició.

2.5 COMPARATIVA DE FRAMEWORKS

Els Framework que hem estudiat son tots molt diferents i de cada un se'n pot extreure coses positives de cara a la implementació del nostre.

Spring es un Framework mot potent que contempla tota la vida d'un programa J2EE, no només la implementació del patró MVC. Totes les funcionalitats addicionals, encara que mot àgils, no ens interessen en aquest projecte.

Struts es la solució inicial i la més utilitzada. Es basa en un Servlet que fa de controlador i en fitxers XML per controlar el flux de navegació. També es basa en accions per encapsular la lògica de negoci, i cada acció té un formulari que conté les dades i l'estat.

La part de Spring Web que implementa el patró MVC es semblant a Struts i utilitza també un Servlet com a controlador, però té coses interessants que el diferencien de Struts: pot tenir més d'un Servlet que faci de controlador, el concepte de programació basada en anotacions per controlar el flux de navegació i pot utilitzar més tipus de vista a part de JSP.

Struts2 utilitza també com Struts fitxers XML per controlar el flux de navegació i accions per connectar-se a la lògica de negoci real, però té diferències significatives amb Struts: El controlador no es un Servlet sinó que es un Filtre, les accions no cal que estenguin cap classe ni implementin cap interfície, només cal que tingui un mètode execute(), pot utilitzar anotacions per substituir fitxers XML, utilitza interceptors que implementen diferents capes de funcionalitat independent de la acció com pot ser control de flux, validació de dades, seguretat, etc.

JSF es el més diferent i es basa en components que tenen el seu propi estat, els seus validadors i els seus convertidors de dades, a més crea les seves pròpies etiquetes per manejar-los. Aquests components representen parts de les vistes i es comuniquen directament amb el servidor. Aquests components no implementen ells sols el patró MVC, JSF necessita un controlador que implementa amb un Servlet. Aquest Servlet pot ser substituït sense problemes per Struts, així es pot aprofitar la potencia d'ambdós.

Tots els Frameworks estudiats creen les seves pròpies etiquetes per facilitar-li la seva tasca al dissenyador de la pàgina i evitar ficar codi Java.

2.6 SELECCIÓ DE LES CARACTERÍSTIQUES DEL NOSTRE FRAMEWORK

Utilitzarem Filtres per implementar el controlador, ja que, aporta un mecanisme molt interessant per incloure controls abans i després d'executar la acció.

Per mapejar les accions que hem de cridar ho farem per nom, el fet que cada esdeveniment cridi a la seva acció i que les accions es diguin igual que les crides estalvia molta escriptura de XML i no perd gens de potència ni agilitat. Cada acció tindrà el seu formulari com Struts que també es mapejarà per nom.

La utilització d'anotacions per controlar el flux de navegació té el desavantatge que s'ha d'escriure en el propi font del controlador o de les accions, havent de modificar el codi per variar la navegació. Els fitxers XML tenen com a desavantatge que s'han d'escriure i quadrar-los amb els resultats de les accions. Per implementar el flux de navegació utilitzarem els propis formularis.

Els formularis seran unes classes en les que es definiran els camps d'entrada i de sortida i mitjançant unes anotacions especials es definirà: si els camps d'entrada son opcionals o obligatoris, les regles de validació dels camps d'entrada, els possibles estats dels camps de sortida segons valors, estat general del formulari depenent dels estats individuals de cada camp i el formulari següent on navegar depenent de l'estat general. Cada acció rebrà el seu formulari amb els camps d'entrada i el retornarà amb els camps de sortida sense fer cap validació.

El controlador cridarà primer una classe Mapping que passant-li el HttpRequest i el nom del formulari s'encarregarà d'omplir i validar els camps d'entrada i si hi ha algun error cridarà al dispatcher el mateix formulari amb els errors carregats. Després cridarà a la acció passant-li el formulari que rebrà ple amb les dades de sortida. Després cridarà a una classe Resolver que amb el formulari ple determinarà quina es la propera pàgina a presentar. I finalment cridarà al Dispatcher que enviarà la pàgina HTTP creada al cient.

Utilitzarem JSP i etiquetes JSP per crear les vistes. També crearem les nostres pròpies etiquetes per ajudar al dissenyador de la pàgina. Aquestes etiquetes s'utilitzaran per implementar parts repetitives a les pàgines com mostrar errors, visualitzar taules, etc.

3 - CONSTRUCCIÓ DEL FRAMEWORK

3.1 ANÀLISI

El Framework que volem dissenyar s'ha de poder utilitzar en qualsevol entorn i per qualsevol aplicació J2EE, però estem pensant en un Framework que doni moltes facilitats per implementar programari J2EE per aplicacions executades en intranets d'empresa en les que cada operari connectat al programa pot estar en diferents estats i al connectar-se ha de saber recuperar l'estat en el que està. Ha de complir els següents requisits:

- Al executar una acció un usuari, s'ha de poder recobrar l'estat en el que està. Per tant, una acció pot donar com a resultat diferents estats que poden fer navegar a pàgines diferents.
- La detecció d'errors en les dades introduïdes s'han de poder processar sense haver d'accedir novament a la base de dades per obtenir la informació que tenim a la pantalla.

Tal com hem definit en el apartat Selecció de les característiques del Framework la peça principal del Framework seran els formularis. Hi haurà dos tipus de formularis:

- ActionForms que els utilitzarà la acció i contindran els paràmetres de la acció i totes les possibles dades, classes i llistes que pot utilitzar la acció. S'encarregarà de validar els paràmetres introduïts i depenent dels resultats de la acció resoldrà quina es la propera vista a mostrar.
- JSPForms que els utilitzaran els JSP per recollir les dades que necessita. Tindran un mecanisme que automàticament carregarà les dades des del actionForm.

Aquests formularis seran molt senzills de construir, s'han de definir simplement les classes i els paràmetres que el composen, les validacions dels paràmetres i el mecanisme de resolució de la propera vista.

Tindrem un mecanisme de reconstrucció de formularis per poder tornar al mateix JSP si es detecta un error sense haver de tornar a demanar les dades. Si es detecta un error validant les dades introduïdes tornarà al JSP original sense comprovar cap més estat.

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

Fora de l'abast d'aquest projecte, al reconstruir un formulari inicialitzarem els paràmetres d'entrada del formulari amb els camps introduïts per l'usuari.

El disseny general del Framework serà el següent.

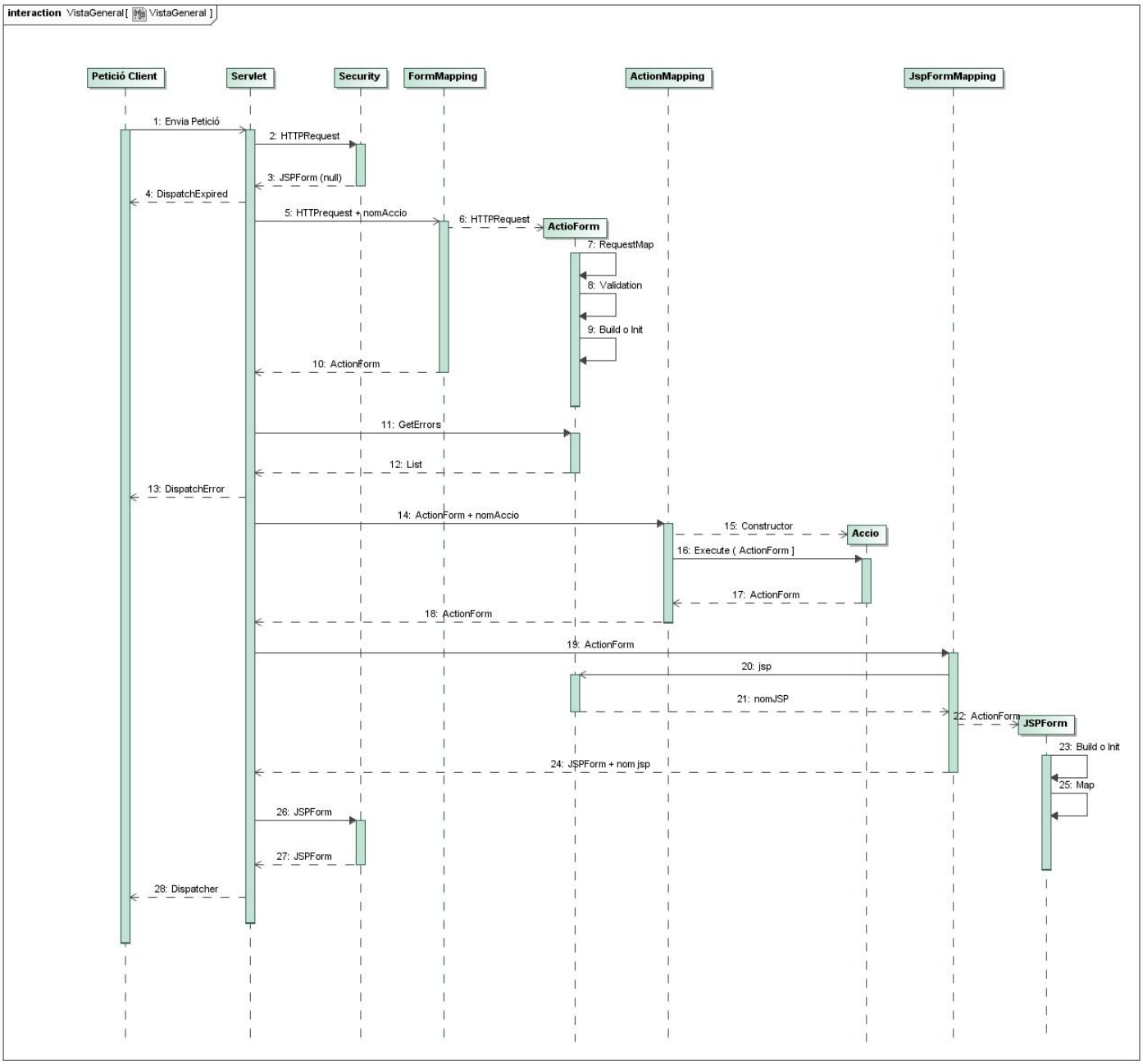


Figura 11: Diagrama de seqüència del flux general del Framework

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

Un Servlet únic rebrà les peticions de l'usuari i s'encarregarà de tornar-li la resposta.

Una classe security es guardarà les pàgines enviades per tornar-les reconstruïdes, també controlarà la caducitat de les pàgines.

Una classe FormMapping s'encarregarà de crear el formularis i de guardar-los per si es tornen a demanar posteriorment, en aquest cas s'inicialitzaran. Cada formulari estendrà una classe ActionForm estàndard que ja té implementat el constructor(HttpRequest) que construirà automàticament cada formulari a partir de es anotacions fetes pel desenvolupador i carregarà i validarà automàticament tots els camps d'entrada des del HTTPrequest. Aquesta classe genèrica que representa tots els formularis d'acció tindrà implementat també un mètode perquè el Servlet pugui recollir els errors de validació i un altre per resoldre la propera vista.

El desenvolupador només haurà de crear els nomaccioForm que estendrà ActionForm fent simplement unes anotacions en les que definirà els paràmetres, classes i llistes que ha de contenir, les regles de validació dels paràmetres si n'hi ha i les regles de resolució de la propera vista. No s'hi haurà de ficar cap mena codi i seran molt senzills de construir.

Una classe ActionMapping s'encarregarà de crear les accions i guardar-les per poder-les executar altres vegades. També s'encarregarà d'executar la acció i retornar el formulari ple.

Una classe JspFormMapping s'encarregarà de demanar al ActionForm que li resolgui el JSP que s'ha de servir, de crear el JspForm i de guardar-lo per si s'ha de demanar posteriorment, en aquest cas s'inicialitzarà. Cada formulari d'aquests estén una classe JspForm estàndard que s'encarregarà de construir els camps dels formularis a partir d'anotacions similars a les dels ActionForms i s'encarregarà de traspassar automàticament els camps de l'actionFrom al JspForm. Els JspForm seran més senzills perquè no necessiten paràmetres d'entrada i tindran molts menys camps de sortida perquè si cada acció pot navegar a diferents formularis haurà de contenir els camps de tots. Aquesta diferenciació separa més la construcció de pàgines i d'accions. El JspFormMapping retornarà el formulari JSP i el nom del jsp al Servlet perquè s'encarregui d'enviar-lo.

3.2 Disseny

3.2.1 Estudi dels patrons a utilitzar

3.2.1.1 MVC

El patró de disseny es la base d'aquest Framework que es basa en separar la vista del model perquè els desenvolupadors de la vista puguin fer les seves modificacions sense implicar el model i els desenvolupadors del model puguin implementar-lo i millorar-lo sense haver de pensar en la vista.

En el nostre Framework:

- La vista ha de desenvolupar els JSP i les anotacions en els JspForm, per tant, son absolutament independents de la vista i del controlador.
- El model desenvolupa les accions omplint els actionForms corresponents a cada estat de la acció.
- El controlador son anotacions en el ActionFrom que s'encarreguen de resoldre el flux de la aplicació.

Per tant el patró de disseny MVC queda perfectament clar.

3.2.1.2 INTERCEPTING FILTER

El patró Intercepting Filter es molt interessant i l'implementarem en el nostre Framework, però en aquest projecte només utilitzarem el filtre Security que es veu a la figura 11. Aquest serà el primer filtre per defecte, més endavant se'n podran afegir d'altres.

Amb el patró Intercepting Filter s'intercepta i es manipula una petició abans i després que aquesta sigui processada i s'aconsegueix centralitzar els processos comuns a totes les sol·licituds.

Per exemple: es poden implementar filtres per fer autentificacions, conversió d'imatges, Comprimir i descomprimir dades, Encriptar i desencriptar la informació, etc.

Implementarem aquest patró segons els següents diagrames:

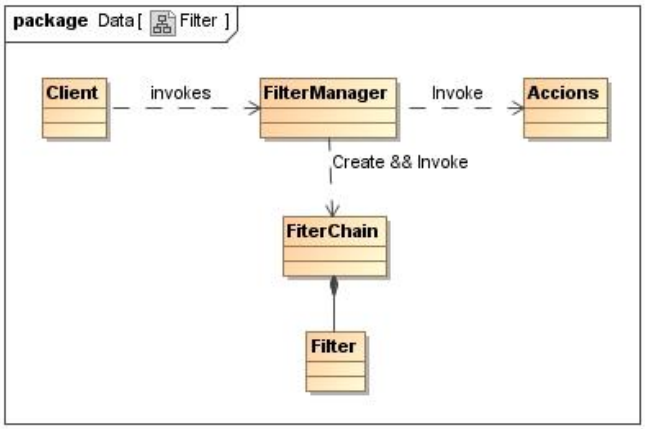


Figura 12: Diagrama de classes del patró Intercepting Filter

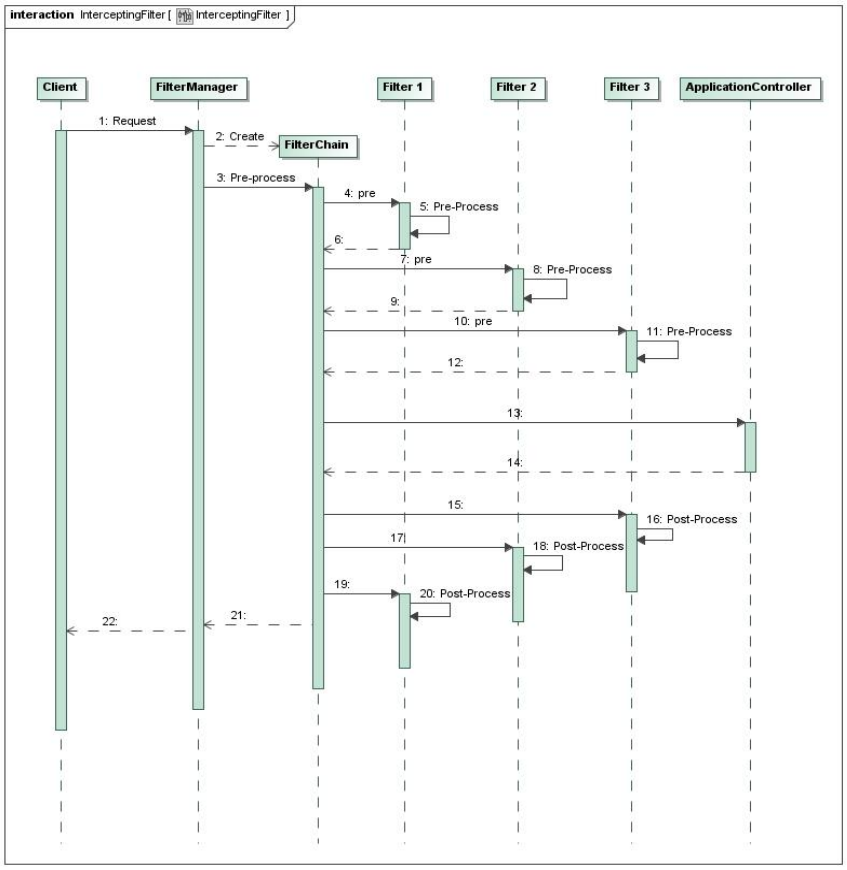


Figura 13: Diagrama de seqüències del patró Intercepting Filter

Intercepting Filter actua com una Pipe on FiterChain fa de túnel. Dins de cada Filtre es crida a FilterChain a mig procés perquè pugi un nivell més. Quan no

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

troba cap nivell més crida el applicationController que s'encarrega de tot el procediment per executar la acció corresponent.

3.2.1.3 Front Controller

El patró Front Controller s'utilitza per centralitzar el maneig de peticions des de la capa de presentació. Amb això s'evita un control duplicat i s'aconsegueix aplicar una lògica comuna a diverses sol·licituds.

Per implementar-lo s'utilitza un controlador central que centralitza la lògica de i gestiona les activitats. Per nosaltres el front controller serà el mateix servlet.

Utilitzarem la estratègia Front Controller junt amb la estratègia Comandament i implementarem aquest patró segons els següents diagrames:

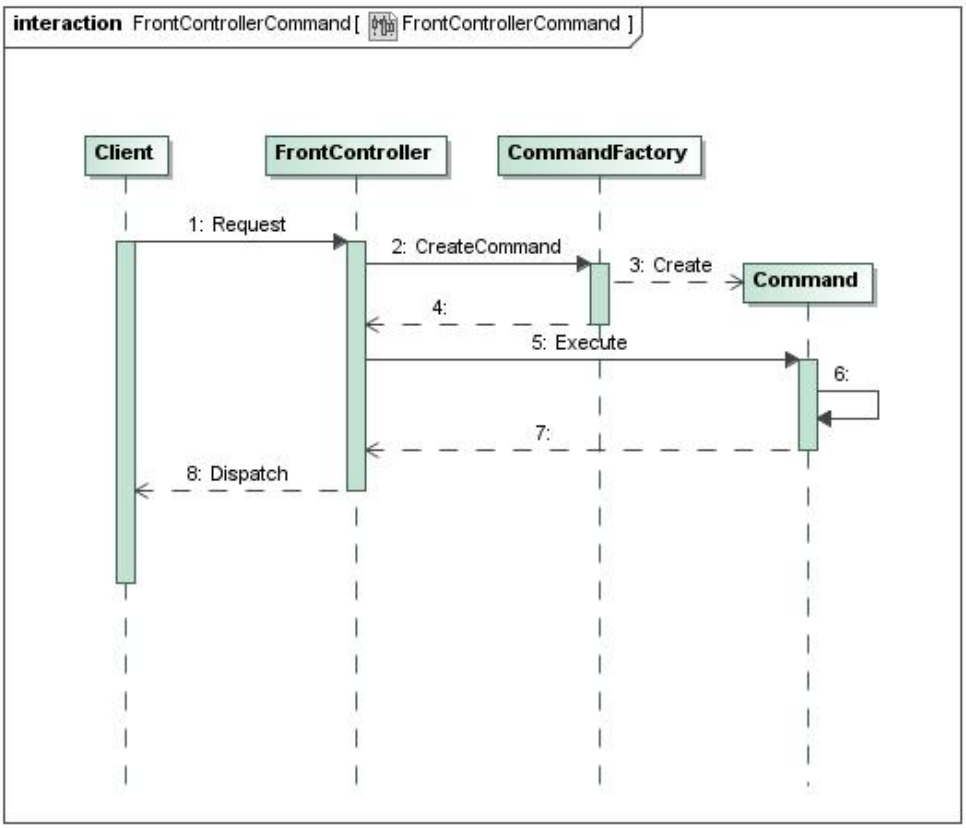


Figura 14: Diagrama de seqüències del patró FrontController i Command

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

3.2.1.4 VIEW HELPER

Aquest patró s'utilitza per separar la vista del procés de negoci evitant posar codi dins les vistes JSP. Per implementar-lo s'utilitzen els Helper per encapsular lògica de negoci dins la vista. Aquests Helper s'implementen com a etiquetes personalitzades. L'utilitzarem tant amb les etiquetes JSP estàndard com amb les pròpies.

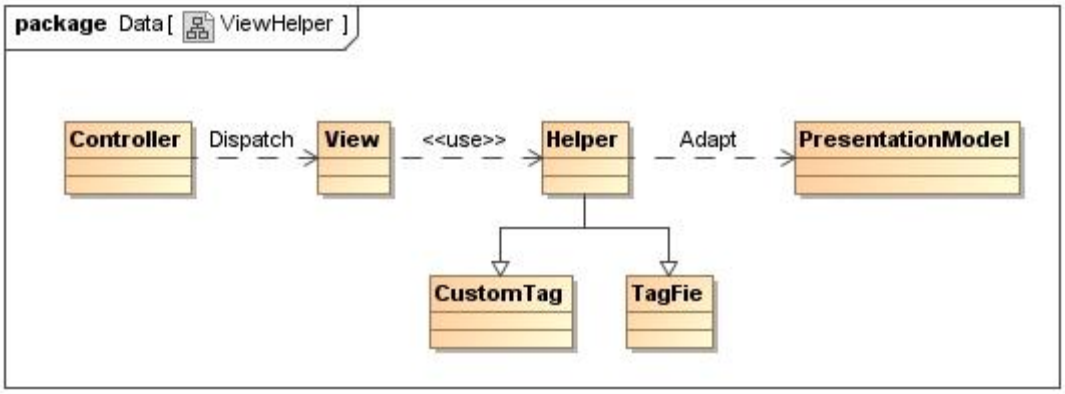


Figura 15: Diagrama de classes del patró View Helper

Utilitzarem el mecanisme estàndard dels JSP i que s'adapta a aquest patró.

3.2.1.5 Service to Worker

S'invoca a la lògica de negoci abans de passar el control a la vista (Que utilitza el Helper que hem vist en el punt anterior). Aquest patró implementa el funcionament general de tots els Frameworks que hem estudiat: Primer crida a la lògica de negoci mitjançant les accions i després es passa el control a la vista.

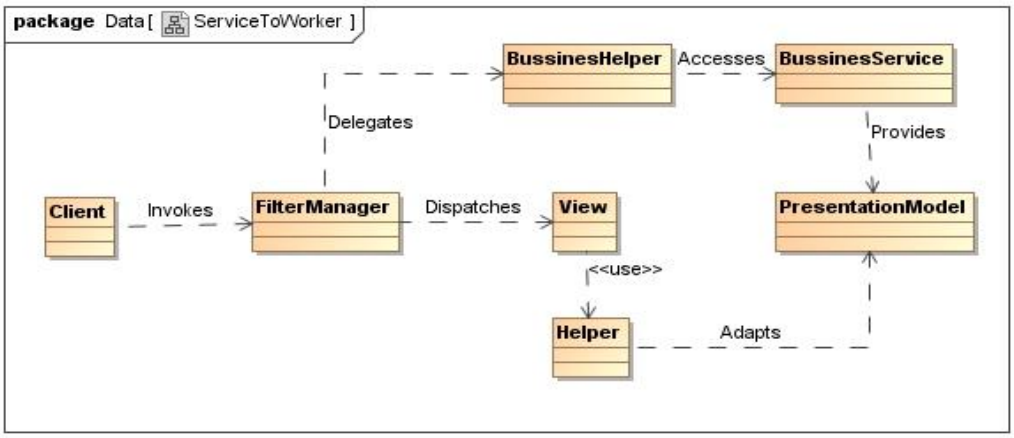
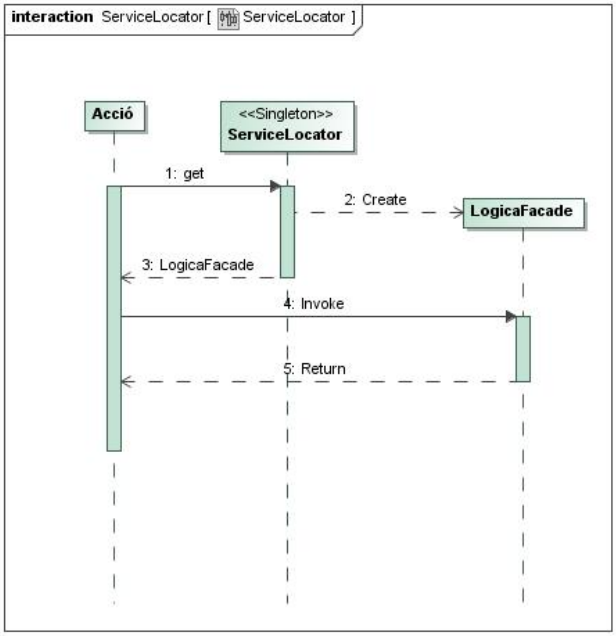


Figura 16: Diagrama de classes del patró Service to Worker

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

3.2.1.6 SERVICE LOCATOR

Utilitzarem el patró Service Locator juntament amb el patró FrontControler per facilitar l'accés de les accions a la lògica real de negoci. Els serveis de la lògica de negoci estaran tots en una classe FrontController que implementarà una interfície i aquesta serà creada i guardada pel servei de localització. Cada cop que es necessiti el serviceLocator la oferirà.



Aquest es el diagrama que implementa el patró Service Locator:

Figura 17: Diagrama de seqüències del patró Service Locator

3.2.1.7 Factory

El patró de disseny Factory l'utilitzarem per crear i guardar les diferents classes: action, actionForm i JspForm. Les classes de la figura 11 ActionMapping, FormMapping i JspFormMapping utilitzaran aquest patró.

El patró Factory s'utilitza per crear classes de la mateixa família d'objectes sense especificar la seva classe concreta. Aquesta família ve representada normalment per una interfície, en aquest projecte utilitzarem una interfície en el cas ActionMapping, però utilitzarem una classe genèrica abstracta en els casos FormMapping i JspFormMapping que ens servirà igual però li podrem donar funcionalitat comuna.

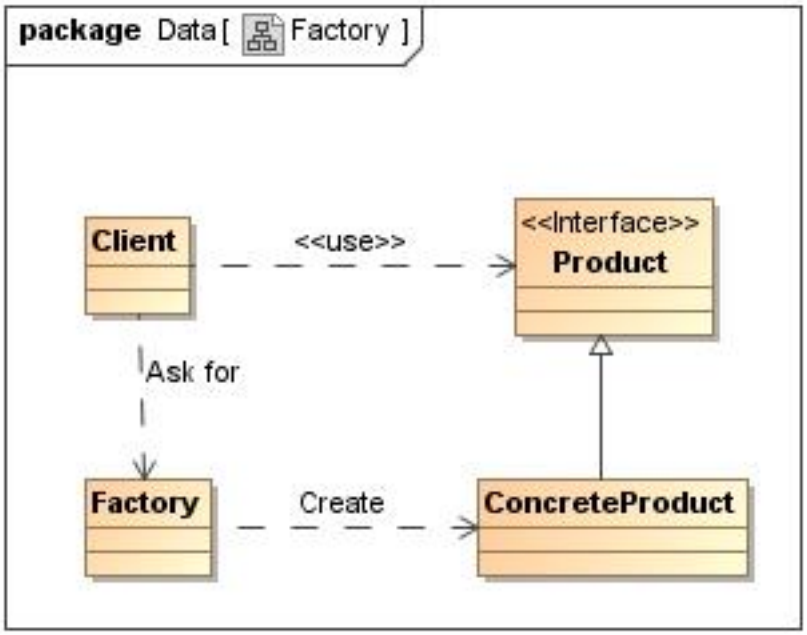


Figura 18: Diagrama de classes del patró Factory

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

3.2.2 DISSENY

Després de decidir els patrons de disseny que utilitzarem, passarem a definir la estructura definitiva del nostre Framework partint del diagrama d'anàlisi de la figura 11. A més de complir amb els requisits d'anàlisi descrits i implementar el patró Model vista controlador, que era un requisit inicial, aquest Framework ha d'usar els patrons estudiats per fer-lo clar, facilitar el manteniment i no limitar el seu creixement.

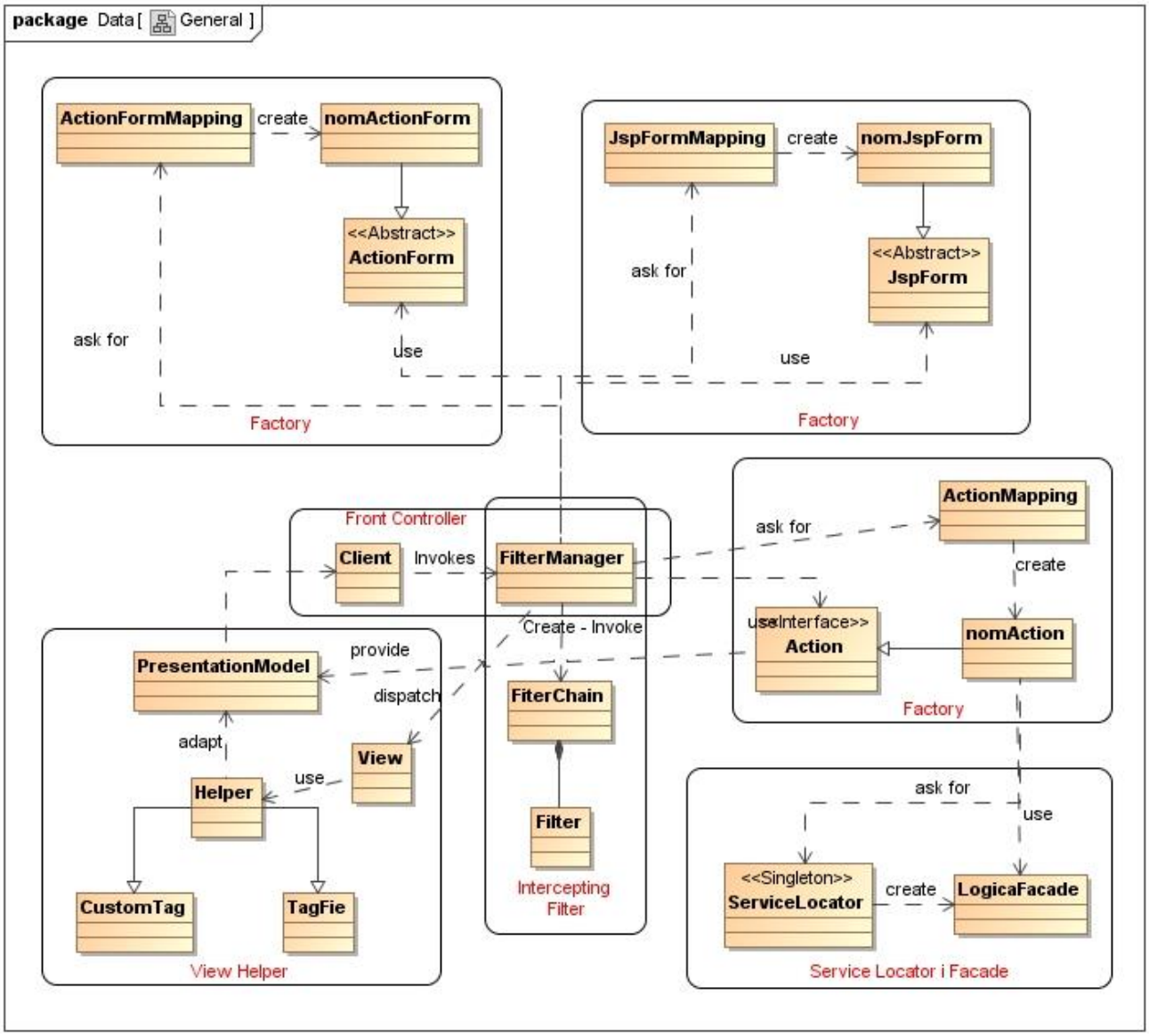


Figura 19: Diagrama de classes del Framework

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

El patró Service To Worker no està anotat en la figura 17 perquè descriu una combinació d'altres patrons i es el fet de combinar el controlador amb un dispatcher, vistes i helpers.

Aquest Framework no necessita cap fitxer de configuració especial a part de definir el servlet en l'arxiu web.xml. La configuració es fa en els formularis actionForm i es la part més complexa del framework. El fet de ficar la configuració de la navegació en els formularis encapsula molt el problema doncs la definició de les dades, dels estats segons els resultats de les dades i el formulari següent depenent de l'estat es fa al mateix lloc. El programador haurà d'implementar les accions i el actionForm i el dissenyador els formularis JSP i els JspActionForm. Els JspActionForm seran molt fàcil d'implementar, ja que només cal enumerar les dades que ha de recollir del actionForm que el cridi.

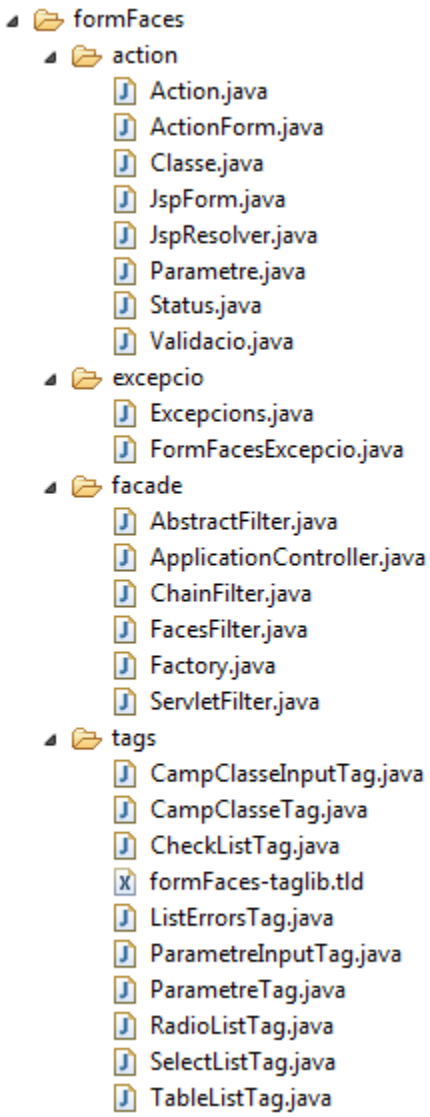
Es crearan uns errors especials que aniran recollint els errors de validació i de procés.

A part de tota la lògica dels actionForms, s'hauran de crear etiquetes per tractar-los en els Jsp de manera que sigui totalment transparent per al desenvolupador de les pàgines, la manera en que estan construïts els formularis. També es crearà una etiqueta que mostri els errors recollits de procés.

Al Framework l'anomenarem formFaces. Form perquè el controlador està basat en formulari iFaces perquè vol implementar la part de validació del JavaServerFaces en la que si es detecta un error en les dades introduïdes retorna els errors al mateix formulari sense haver de cridar a la lògica de negoci.

3.3 Programació

3.3.1 Estructura de classes



Aquest Framework està dividit en quatre parts:

- Servlet i filtre
- Accions i formularis
- Excepcions
- Etiquetes.

3.3.2 EXCEPCIONS

La classe FormFacesExcepcio estén la classe Exception per poder-la utilitzar així en amb el throw, incorpora la gravetat de l'error per poder triar fins quin nivell d'errors mostrem i implementa la interfície Comparable per poder-la classificar per gravetat de l'error:

```
import java.lang.Exception;
/**
* @author Jaume
* Excepció amb gravetat. gravetats possibles
 *        5    Errors definició formularis
 *        4    Errors de procés
 *        3    Errors de validació
 *        2    Warning avís
 *        1    Informatius
 */
public class FormFacesExcepcio extends Exception implements Comparable<FormFacesExcepcio>{

  private int gravetat;

  public FormFacesExcepcio(){
     super();
     gravetat = 0;
  }
  public FormFacesExcepcio(String missatge, int gravetat){
     super(missatge);
     this.gravetat = gravetat;
  }

  public int compareTo(FormFacesExcepcio f){
     if (gravetat==f.getGravetat()){
        return this.getMessage().compareTo(f.getMessage());
     } else {
        return gravetat - f.getGravetat();
     }
  } }
```

I la classe Excepcions que implementa una llista de FormFacesExcepcio i s'encarrega del tractament de la gravetat:

```
public class Excepcions {
  private List<FormFacesExcepcio> excepcions;

  public Excepcions(){
     excepcions = new ArrayList<FormFacesExcepcio>();
  }

  public void nova(FormFacesExcepcio e){
     excepcions.add(e);
  }
```

```
  public List<FormFacesExcepcio> getExcepcions() {
     return excepcions;
  }
  public List<FormFacesExcepcio> getExcepcions(int gravetatMinima) {
     List<FormFacesExcepcio> ex = new ArrayList<FormFacesExcepcio>();
     for (Iterator<FormFacesExcepcio> it=excepcions.iterator(); it.hasNext();){
        FormFacesExcepcio e = it.next();
        if (e.getGravetat()>=gravetatMinima){
           ex.add(e);
        }
     }
     return ex;
  }
  public void setExcepcions(List<FormFacesExcepcio> excepcions) {
     this.excepcions = excepcions;
  }
  public Iterator<FormFacesExcepcio> getItExcepcions() {
     return excepcions.iterator();
  }
  public Iterator<FormFacesExcepcio> getItExcepcions(int gravetatMinima) {
     List<FormFacesExcepcio> ex = getExcepcions(gravetatMinima);
     Collections.sort(ex);
     return ex.iterator();
  }
  public boolean teExcepcions(){
     return !excepcions.isEmpty();
  }
  public boolean teExcepcions(int gravetatMinima){
     List<FormFacesExcepcio> ex = getExcepcions(gravetatMinima);
     return !ex.isEmpty();
  }
}
```

3.3.3 Servlet i Filtre

El funcionament del servlet i del filtre s'explica a la figura 13. El servlet delega al filterChain la gestió de la acció i rep d'ell e nom del jsp i el JspForm dins el request. Amb aquestes dues dades fa un fordward amb el dispatcher. En el constructor es creen els filtres i es guarden per passar-li al filterChain, així no cal crear-los cada vegada.

En aquest projecte només es carregarà el facesFilter que serà sempre el filtre 1 que s'encarrega de recuperar el JspForm anterior amb les dades per poder-lo tornar a enviar. Per la resta de filtres, específics de cada aplicació, que es vulguin crear, s'anotaran en el fitxer XML (filters.XML) . El constructor del ServletFilter els crearà i guardarà.

```
public class ServletFilter extends HttpServlet {
  private List<AbstractFilter> filters;

/**
 * Al crear el ServletFilter es creen tots els filtres així al crear la nova pipe ChainFilter
 * se li poden passar tots creats i no cal crear-los cada vegada
 */
  public ServletFilter() {
     super();
     filters = new ArrayList<AbstractFilter>();
     filters.add(new FacesFilter());
     // Afegir nous fiters des d'un XML
  }

/**
 * Crea una ChainFilter nova cada vegada perquè així cada ChainFilter pot portar una iteració
 * de filtres diferent per cada crida.
 */
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException,IOException {
String nextPage = null;
     try {
        ChainFilter chf= new ChainFilter(filters);
        chf.doFilter(request, response);
        nextPage = chf.getNextPage();
     } catch (Throwable t) {
        t.printStackTrace();
     }
         RequestDispatcher rd =
   getServletContext().getRequestDispatcher(response.encodeURL(nextPage));
     rd.forward(request, response);
  }
  protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
     doGet(req, resp);
  }
}
```

El chainFilter s'encarrega de cridar el següent filtre de la cadena cada vegada que el criden a ell. Si no queda cap filtre més a la cadena, crida al ApplicationController que executa la lògica de la acció.

Cada cop que crida un filtre es passa a ell mateix com a paràmetre perquè el filtre que crida el pugui tornar a invocar.

El ChainFilter es guarda la pàgina següent perquè la recuperi el servlet. Normalment aquesta dada li retorna ApplicationControler després d'executar la lògica de la acció, però FacesFilter li dóna l'actual per si el ActionForm detecta algun error de validació en les dades introduïdes.

```
public class ChainFilter implements FilterChain {
  private List<AbstractFilter> filters;
  private Iterator<AbstractFilter> itFilters;
  private String nextPage;
  public ChainFilter(List<AbstractFilter> flt){
     itFilters = null;
     filters = flt;
     nextPage = null;
  }
  public boolean hasNext(){
     if (itFilters == null){
        return false;
     } else {
        return itFilters.hasNext();
     }
  }
  public String getNextPage() {
     return nextPage;
  }
  public void setNextPage(String nextPage) {
     this.nextPage = nextPage;
  }
  @Override
  public void doFilter(ServletRequest rq, ServletResponse rp)
        throws IOException, ServletException {
     if (itFilters == null){
        itFilters = filters.iterator();
     }
     if (hasNext()){
        itFilters.next().doFilter(rq, rp, this);
     } else {
        Page = (new ApplicationController()).execute(rq, rp);
        // si Page=null es que s'ha detectat un error de validació i s'utilitza el nom de
        // pàquina que ha passat FacesFilter
        if (Page!=null){nextPage=Page;}
     }
  }
}
```

Els filtres han d'estendre la classe AbstractFilter que a la seva vagada implementa la classe Filter. Tots els filtres han d'implementar els mètodes abstractes doPreProcessing() i do postProcessing(). El main processing ja està implementat en AbstractFilter i s'encarrega de cridar de nou a la cadena perquè executi el filtre següent. Amb aquest sistema s'executaran primer tots els preProcessing i després tots es postProcessing en ordre invers.

```
public abstract class AbstractFilter implements Filter {
  private String nextPage;
  @Override
  public void destroy() {
  }
  @Override
  public void doFilter(ServletRequest rq, ServletResponse rp,
        FilterChain fch) throws IOException, ServletException {
     try {doPreProcessing(rq);
     } catch (IOException e) {throw e;
     } catch (ServletException e) {throw e;
     }
       doMainProcessing(rq, rp, fch);
       doPostProcessing(rq);
}
@Override
public void init(FilterConfig fc) throws ServletException {}
public abstract void doPreProcessing(ServletRequest request) throws IOException,
ServletException;
public abstract void doPostProcessing(ServletRequest request);
public void doMainProcessing(ServletRequest request, ServletResponse response, FilterChain
chain){
  // si el filtre ha actualitzat nextPage en el preProcessing, li passa aqui al chainFilter
  if (nextPage !=null){
     chain.setNextPage(nextPage);
  }
  try {
     chain.doFilter(request, response);
}  catch (Exception e){
     System.out.println("Error executant següent doFilter. Filtre: "+this.getClass().getName());}
} }
```

FacesFilter es sempre el primer filtre i s'encarrega de recuperar el JspForm que es va enviar l'últim cop i posar-lo dins el request al començar en el preProcessing(). En el preProcessing() també es guarda la pàgina per guardar-la a ChainFilter en el mainProcessing. En el postProcessing() guarda el JspForm per si s'ha de reutilitzar el proper cop:

```
public class FacesFilter extends AbstractFilter {
  private Hashtable<String, JspForm > formularis;
  @Override
  public void doPreProcessing(ServletRequest request) throws IOException,
        ServletException {
     String idForm = request.getParameter("idForm");
     String nom = "form_faces.jspForms."+idForm+"Form";
     super.setNetPage("/jsp/"+idForm+".jsp");
     if (nom !=null && formularis.containsKey(nom)){
        request.setAttribute("JspForm", formularis.get(nom));
     } else {
        request.setAttribute("JspForm", null);
     }
  }
```


Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

```
@Override
public void doPostProcessing(ServletRequest request) {
   JspForm form = (JspForm)request.getAttribute("JspForm");
   if (form!=null){
      String nom = form.getClass().getName();
      formularis.put(nom, form);
   }
}}
```

ApplicationController s'encarrega d'executar la acció seguint els següents passos:

- Recupera el actionForm corresponent a la acció
- Fa que el actionForm es carregui i validi els paràmetres que porta el request si la validació es incorrecta recupera els errors i acaba. El JspForm i la pàgina següent que ha recuperat el FacesFilter serà la bona.
- Recupera la acció.
- Executa la acció.
- Fa que el actionForm resolgui quina es la propera pàgina.
- Recupera el JspForm corresponent a la propera pàgina
- Fa que el JspForm recuperi els paràmetres i les classes del actionForm.
- Retorna la pàgina següent i el JspForm dins el request.

```
public class ApplicationController {

protected String execute(ServletRequest request, ServletResponse response)
      throws ServletException,IOException {
   String nextPage = null;
   String nameAction = request.getParameter("action");
   ActionForm af = Factory.getReference().getActionForm(nameAction);
   if (af != null){
      af.map(request);
      if (!af.getExcepcions().teExcepcions(3)){
         Action a = Factory.getReference().getAction(nameAction);
         if (a == null){
            af.novaExcepcio(new FormFacesExcepcio( "Acció "+nameAction+" no creada", 5));
            printErrors(af);
         } else {
            try {
               a.execute(af);
               String pageNext = af.resolJsp();
               if (pageNext!=null){
                  JspForm jf = Factory.getReference().getJspForm(af.resolJsp());
                  if (jf != null) {
                     jf.map(af);
```

```
                    if (!af.getExcepcions().teExcepcions(4)){
                        request.setAttribute("JspForm", jf);
                        nextPage = "/jsp/"+pageNext+".jsp";
                    }
                } else {
                    af.novaExcepcio(new FormFacesExcepcio("JspForm "+pageNext+" no creada", 5));
                    printErrors(af);
                }
            } else {
                af.novaExcepcio(new FormFacesExcepcio("Error al resoldre l pàgina JSP de la Acció
                "+nameAction, 4));
                printErrors(af);
            }
        } catch (Exception e){
            af.novaExcepcio(new FormFacesExcepcio("Error al executar Acció "+nameAction+":
            "+e.getMessage(), 4));
            printErrors(af);
        }
    }
    request.setAttribute("excepcions", af.getExcepcions());
}
return nextPage;
}
}
```

Factory es un singleton que es guarda totes les accions, els actionForm i els JspForm perquè així no cal construir-los cada cop.

3.3.4 ACCIONS I FORMULARIS

Els ActionForm i JspForm son la base del Framework. Per poder automatitzar les funcionalitats, no podem utilitzar classes normals i per això hem hagut de crear una classe especial "Classe". Això te l'inconvenient que en les accions s'han d'utilitzar classes d'aquest tipus o passar els atributs a última hora. Aquest inconvenient l'haurem d'arranjar i en properes versions utilitzar classes normals dins els formularis.

El diagrama de classes dels formularis es el següent:

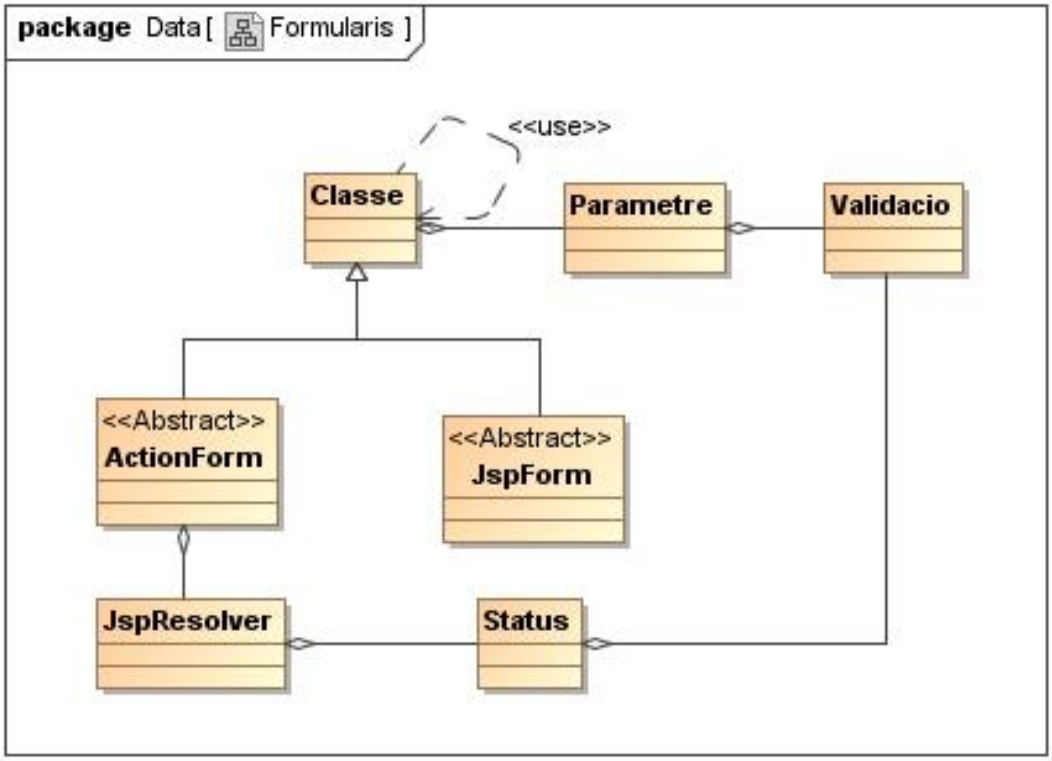


Figura 20: Diagrama de classes dels formularis

La classe Classe conté atributs, altres classes i llistes de classes:

```
private Hashtable<String, Parametre > parametres;
private Hashtable<String, Classe > classes;
private Hashtable<String, List<Classe> > llistes;
```

Un Paràmetre es una classe que conté un paràmetre i una validació:

```
private Comparable parametre;
private String tipus;
private Validacio validacio;
```

Aquests paràmetres poden ser del tipus: String, Integer, Float, Long, Double, Date i Boolean. I quan se li passa un valor el transforma en el tipus corresponent:

```
  public void setValor(String s) throws FormFacesExcepcio{
    try {
      if (tipus.equalsIgnoreCase("String")){
        parametre = s;
      }
      if (tipus.equalsIgnoreCase("Integer")){
        parametre = new Integer(s);
      }
      if (tipus.equalsIgnoreCase("Float")){
        parametre = new Float(s);
      }
      if (tipus.equalsIgnoreCase("Long")){
        parametre = new Long(s);
      }
      if (tipus.equalsIgnoreCase("Double")){
        parametre = new Double(s);
      }
      if (tipus.equalsIgnoreCase("Date")){
        parametre = new Date((new Integer(s.substring(0, 4))).intValue()-1900, (new
Integer(s.substring(4, 6))).intValue()-1, (new Integer(s.substring(6, 8))).intValue());
      }
      if (tipus.equalsIgnoreCase("Boolean")){
        parametre = new Boolean(s);
      }
      if (validacio != null){
        FormFacesExcepcio e = validacio.valida(parametre);
        if (e != null){
          throw e;
        }
      }
    } catch (FormFacesExcepcio e){
      throw e;
    } catch (Exception e){
      throw new FormFacesExcepcio("el valor "+s+" passat no es del tipus "+tipus, 3);
    }
  }
```

Una Validacio es una classe que conté una comparació i una llista de valors a comparar i valida si el paràmetre es valida correctament segons aquesta comparació. Una comparació pot ser del tipus:

- "lt", "gt", "le", "ge", "eq" o "ne" i un valor
- "bt", "nb" (not between) i dos valors
- "in", "ni" (not in) i una llista de valors

```
private String comparacio;
List<Comparable> valors;

public FormFacesExcepcio valida(Comparable element){
   boolean correcte = false;
   String strValors = "Té introduida una comparació mal construïda. Parlar amb el programador.";
   int gravetat = 5;

   if (comparacio.equalsIgnoreCase("lt") && valors.size() == 1){
      correcte = (element.compareTo(valors.get(0))<0);
      strValors = "Ha de ser menor que "+valors.get(0).toString();
      gravetat = 3;
   }
   if (comparacio.equalsIgnoreCase("le") && valors.size() == 1){
      correcte = (element.compareTo(valors.get(0))<=0);
      strValors = "Ha de ser menor o igual que "+valors.get(0).toString();
      gravetat = 3;
   }
   if (comparacio.equalsIgnoreCase("gt") && valors.size() == 1){
      correcte = (element.compareTo(valors.get(0))>0);
      strValors = "Ha de ser mes gran que "+valors.get(0).toString();
      gravetat = 3;
   }
   if (comparacio.equalsIgnoreCase("ge") && valors.size() == 1){
      correcte = (element.compareTo(valors.get(0))>=0);
      strValors = "Ha de ser mes gran o igual que "+valors.get(0).toString();
      gravetat = 3;
   }
   if (comparacio.equalsIgnoreCase("eq") && valors.size() == 1){
      correcte = (element.compareTo(valors.get(0))==0);
      strValors = "Ha de ser igual a "+valors.get(0).toString();
      gravetat = 3;
   }
   if (comparacio.equalsIgnoreCase("ne") && valors.size() == 1){
      correcte = (element.compareTo(valors.get(0))!=0);
      strValors = "Ha de ser diferent de "+valors.get(0).toString();
      gravetat = 3;
   }
   if (comparacio.equalsIgnoreCase("bt") && valors.size() == 2){
            correcte = (element.compareTo(valors.get(0))>=0)&&
            (element.compareTo(valors.get(1))<=0);
      strValors = "Ha d'estar entre "+valors.get(0).toString()+" i "+valors.get(1).toString();
      gravetat = 3;
   }
   if (comparacio.equalsIgnoreCase("nb") && valors.size() == 2){
            correcte = (element.compareTo(valors.get(0))<0)||
            (element.compareTo(valors.get(1))>0);
            strValors = "No ha d'estar entre "+valors.get(0).toString()+" i "+valors.get(1).toString();
      gravetat = 3;
   }
```


Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

```
        if (comparacio.equalsIgnoreCase("ni")){
            strValors = "";
            gravetat = 3;
            boolean primer = true;
            for (Iterator<Comparable> it = valors.iterator();it.hasNext();){
                Comparable valor = it.next();
                if (primer){
                    strValors = "No pot ser un dels valors: "+valor.toString();
                    primer = false;
                } else {
                    strValors = strValors + ", " + valor.toString();
                }
                if (element.compareTo(valor)==0){
                    correcte = false;
                }
            }
        }

        if (comparacio.equalsIgnoreCase("in") && valors.size() > 0){
            correcte = false;
            strValors = "";
            gravetat = 3;
            boolean primer = true;
            for (Iterator<Comparable> it = valors.iterator();it.hasNext();){
                Comparable valor = it.next();
                if (primer){
                    strValors = "Ha de ser un dels valors: "+valor.toString();
                    primer = false;
                } else {
                    strValors = strValors + ", " + valor.toString();
                }
                if (element.compareTo(valor)==0){ correcte = true; }
            }
        }
        if (correcte){
            return null;
        } else {
            return new FormFacesExcepcio(strValors, gravetat);
        }
    }
```

Un ActionForm es una Classe que conté un mètode per recuperar els paràmetres del request i també conté una llista de JspResolver que es la encarregada de resoldre el proper formulari a mostrar. Per saber el nom del proper formulari el ActionForm segueix aquesta llista de manera seqüencial fins que troba el primer JspResolver que li torna un nom de Jsp.

```
    private List<JspResolver> resolver;

    public void map(ServletRequest request) {
        for (Enumeration<String> par = super.getClauParametres(); par.hasMoreElements();){
            String param = par.nextElement();
            String valor = request.getParameter(param);
            if (valor == null){
                super.novaExcepcio(new FormFacesExcepcio("Parametre passat "+param+" no creat al
formulari", 5));
            } else {
                super.setValor(param, valor);
            }
        }
    }

    public String resolJsp(){
        String jsp = null;
        int i = 0;
        while (jsp==null & i < resolver.size()){
            jsp = resolver.get(i).resolve(this);
            i++;
        }
        return jsp;
    }
```

Un JspResolver es una classe que conté el nom d'un Jsp i una llista d'estats. Si tots els estats son certs el JspResover retorna el nom del Jsp.

```
    private String jsp;
    private List<Status> estats;

    public String resolve(ActionForm a){
        boolean correcte = true;
        for (Iterator<Status> it = estats.iterator(); it.hasNext();){
            if(!it.next().valida(a)){ correcte = false; }
        }
        if (correcte){ return jsp;
        } else { return null; }
    }
```

Un Status es una validació sobre un paràmetre del formulari, sobre un paràmetre d'una classe del formulari o comprova si una llista del formulari té elements. En el constructor, si la validació es fa sobre el paràmetre d'una classe es passa amb un punt de separació: classe.parametre.

```
public class Status {
   private String llista;
   private String classe;
   private String parametre;
   private Validacio validacio;

   public Status(String tipus, String camp) {
      super();
      llista = null;
      classe = null;
      parametre = null;
      validacio = null;
      if (tipus.equalsIgnoreCase("llista")){
         llista = camp;
      }
      if (tipus.equalsIgnoreCase("campClasse")){
         int i = camp.indexOf(".");
         classe = camp.substring(0, i);
         parametre = camp.substring(i+1);
      }
      if (tipus.equalsIgnoreCase("parametre")){
         parametre = camp;
      }
   }

   public boolean valida(ActionForm a){
      boolean estat = true;
      if (llista!=null){
         if (a.getLlista(llista)==null || a.getLlista(llista).isEmpty()){
            estat = false;
         }
      } else {
         if (classe!=null){
            if (a.getClasse(classe)==null ||
               a.getClasse(classe).getValor(parametre)==null){
               estat = false;
            } else {
               if (validacio!=null && validacio.valida
                  (a.getClasse(classe).getParametre(parametre))!=null){
                  estat = false;
               }
            }
         } else {
            if (parametre == null){
               estat = false;
            } else {
               if (validacio!=null && validacio.valida(a.getValor(parametre))!=null){
                  estat = false;
               }
            }
         }
      }
      return estat; }
```


Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

```
    public String getTipusCamp(ActionForm a){
        String tipus = null;
        if (llista!=null){
            if (a.getLlista(llista)!=null){
                tipus = "Llista";
            }
        } else {
            if (classe!=null){
                if (a.getClasse(classe)!=null &&
                    a.getClasse(classe).getTipusParametre(parametre)!=null){
                    tipus = a.getClasse(classe).getTipusParametre(parametre);
                }
            } else {
                if (parametre != null && a.getTipusParametre(parametre)!=null){
                    tipus = a.getTipusParametre(parametre);
                }
            }
        }
        return tipus;
    }
}
```

La Classe es abstracta i qualsevol classe que la estengui ha d'implementar el mètode definicio(). Implementar aquest mètode consisteix en una sèrie de crides a mètodes de la classe que construeixen el formulari. Així la classe Prova:

```
public class ProvaClasse extends Classe {

    @Override
    protected void definicio() {
        addParametre("preu", "Float");
    }
}
```

Construeix una classe amb un sol paràmetre que s'anomenarà preu i que es de tipus Float.

Per definir una classe es poden fer les següents crides:

- addParametre(String, String) crea els paràmetres. El primer String es el nom i el segon es el tipus de parametre que pot ser: String, Integer, Float, Long, Double, Date (Els valors de les dates en format AAAAMMDD), Boolean.

- addValidacio(String) crea una validació per l'últim paràmetre. String es la comparació: le (menor o igual) lt (menor), ge major o igual, gt major, eq igual, eq (igual), ne (diferent), bt (entre dos valors), nb (no està entre dos valors), in (pren un dels valors de la llista), ni (no pren cap dels valors de la llista.
- addValor(Comparable) afegeix un valor a comparar a la última validació
- addValidacio(String, Comparable) crea una validació i li afegeix un valor a comparar
- addValidacio(String, Comparable, Comparable) crea una validació i li afegeix dos valors a comparar
- addClasse(String, String) crea les classes que porten les dades del formulari.
- addList(String, String) crea les llistes de classes que portaran els valors.

Un ActionForm estén una Classe, per tant tot actionForm també haurà d'implementar el mètode definicio(). I també ha d'implementar el mètode definicioResolucioJsp() que té una filosofia molt semblant al mètode definicio():

- addJsp(String) crea un nou JspResolver per resoldre aquest Jsp
- addJspStatus(String, String) crea un nou Status i l'afegeix al JspResolver.
- addValidacioStatus(String) crea una validació per l'últim Status. String es la comparació que està definida a validació
- addValorStatus(Comparable) afegeix un valor a comparar a l'última validació
- addValidacioStatus(String, Comparable) crea una validacio per l'últim status i li afegeix un valor a comparar
- addValidacioStatus(String, Comparable, Comparable) crea una validacio per l'últim status i li afegeix dos valors a comparar

un exemple d'un actionForm seria:

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

```
public class ProvaForm extends ActionForm {

  @Override
  protected void definicio() {
    addParametre("sencer", "Integer");
    addValidacio("lt", 4);
    addParametre("text", "String");
    addValidacio("in", "UN", "DOS");
    addValor("TRES");
    addClasse("formulari","ProvaClasse");
    addList("formularis","ProvaClasse");
  }

  @Override
  protected void definicioResolucioJsp() {
    addJsp("jsp1");
    addJspStatus("llista", "formularis");
    addJsp("jsp2");
    addJspStatus("parametre", "text");
    addValidacioStatus("eq", "DOS");
    addJspStatus("campClasse", "formulari.flotant");
    addValidacioStatus("bt", "3", "5");
    addJsp("jsp3");
  }
```

Es un formulari amb un sencer que ha de ser menor que 4, un text que ha de valer UN, DOS o TRES, una classe provaClasse que hem definit abans i una llista de classes.

La resolució d el següent formulari tornarà “jsp1“ si la llista té elements, en cas contrari tornarà “jsp2” si el parametre text es igual a “DOS” y el camp flotant de la classe formulari pren valors entre 3 i 5.. en cas contrari tornarà “jsp3”.

Un JspForm estén també la una classe, o sigui que s’ha d’implementar el mètode definicio(). A més té implementat un mètode map que passa els paràmetres i classes necessaris des d’un actionForm. Per passar les classes de l’ActionForm al JspForm no utilitza = sinó que utilitza un mètode clone(Classe) implementat en Classe. Es fa així perquè al inicialitzar les classes d’un action form no s’inicialitzin les dels JspForm que han fet map d’aquest ActionForm ja que llavors no serviria de res que se’ls guardés el FacesFilter.

```
public abstract class JspForm extends Classe {

  public void map(ActionForm form) {
        for (Enumeration<String> par = super.getClauParametres();
          par.hasMoreElements();){
          String param = par.nextElement();
          String valor = form.getValor(param);
          if (valor == null){
            form.novaExcepcio(new FormFacesExcepcio("Parametre del JspForm
              "+param+" no existeix al ActionForm", 5));
          } else {super.setValor(param, valor); }
        }

    for (Enumeration<String> cla = super.getClauClasses(); cla.hasMoreElements();){
      String param = cla.nextElement();
      Classe valor = form.getClasse(param);
      if (valor == null){
        form.novaExcepcio(new FormFacesExcepcio("Llista del JspForm
          "+param+" no existeix al ActionForm", 5));
      } else {
        Classe actual = getClasse(param);
        actual.clone(valor);
      }
    }

    for (Enumeration<String> lli = super.getClauLlistes(); lli.hasMoreElements();){
      String param = lli.nextElement();
      List<Classe> valor = form.getLlista(param);
      if (valor == null){
        form.novaExcepcio(new FormFacesExcepcio("Llista del JspForm
          "+param+" no existeix al ActionForm", 5));
      } else {
        List<Classe> actual = this.getLlista(param);
        actual.clear();
        for (Iterator<Classe> it=valor.iterator(); it.hasNext();){
          actual.add(it.next());
        }
      }
    }
    excepcions = form.getExcepcions();
  }

  @Override
  protected abstract void definicio();}
```

Una acció ha d'implementar el mètode execute(ActionForm) que conté la interfície acció

```
public interface Action{
  public void execute(ActionForm af) throws Exception;
  }
```

3.3.5 Etiquetes

Les etiquetes encapsularan el codi per tractar els JspForm sense que el dissenyador de la pàgina sàpiga com estan construïdes les Classe. S'intenta que el dissenyador de pàgines no posi pràcticament gens de codi i es dediqui al disseny. S'han fet les següents etiquetes:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>formFaces</shortname>
  <uri></uri>
  <info>
    etiquetes especials pel framework formFaces
  </info>
  <tag>
    <name>tableList</name>
    <tagclass>formFaces.tags.TableListTag</tagclass>
    <teiclass></teiclass>
    <bodycontent>EMPTY</bodycontent>
    <info>Crea una taula amb els camps d'una llista</info>
    <attribute>
      <name>nomLlista</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>border</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>camps</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>funcio1</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>funcio2</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>funcio3</name>
      <required>false</required>
    </attribute>
  </tag>
```

```
<tag>
  <name>SelectList</name>
  <tagclass>formFaces.tags.SelectListTag</tagclass>
  <teiclass></teiclass>
  <bodycontent>EMPTY</bodycontent>
  <info>Crea una llista de seleccio amb els camps d'una llista</info>
  <attribute>
    <name>nomLlista</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>nomSelect</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>campValor</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>campEtiqueta</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>valorPerDefecte</name>
    <required>false</required>
  </attribute>
</tag>
<tag>
  <name>RadioList</name>
  <tagclass>formFaces.tags.RadioListTag</tagclass>
  <teiclass></teiclass>
  <bodycontent>EMPTY</bodycontent>
  <info>Crea una radio button amb els camps d'una llista</info>
  <attribute>
    <name>nomLlista</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>nomRadio</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>campValor</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>campEtiqueta</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>valorPerDefecte</name>
    <required>false</required>
  </attribute>
</tag>
```

```
<tag>
  <name>CheckList</name>
  <tagclass>formFaces.tags.CheckListTag</tagclass>
  <teiclass></teiclass>
  <bodycontent>EMPTY</bodycontent>
  <info>Crea una check list amb els camps d'una llista</info>
  <attribute>
    <name>nomLlista</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>campValor</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>campEtiqueta</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>campCertFals</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>valorCert</name>
    <required>false</required>
  </attribute>
</tag>
<tag>
  <name>CampClasse</name>
  <tagclass>formFaces.tags.CampClasseTag</tagclass>
  <teiclass></teiclass>
  <bodycontent>EMPTY</bodycontent>
  <info>retorna el valor d'un camp d'una classe</info>
  <attribute>
    <name>nomClasse</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>nomCamp</name>
    <required>true</required>
  </attribute>
</tag>
<tag>
  <name>CampClasseInput</name>
  <tagclass>formFaces.tags.CampClasseInputTag</tagclass>
  <teiclass></teiclass>
  <bodycontent>EMPTY</bodycontent>
  <info>crea un input amb el valor d'un camp d'una classe</info>
  <attribute>
    <name>nomClasse</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>nomCamp</name>
    <required>true</required>
  </attribute>
```


Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

```
    <attribute>
      <name>tipusInput</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>nomInput</name>
      <required>false</required>
    </attribute>
  </tag>
  <tag>
    <name>ParametreInput</name>
    <tagclass>formFaces.tags.ParametreInputTag</tagclass>
    <teiclass></teiclass>
    <bodycontent>EMPTY</bodycontent>
    <info>crea un input amb el valor d'un parametre</info>
    <attribute>
      <name>nomParametre</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>tipusInput</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>nomInput</name>
      <required>false</required>
    </attribute>
  </tag>
  <tag>
    <name>Parametre</name>
    <tagclass>formFaces.tags.ParametreTag</tagclass>
    <teiclass></teiclass>
    <bodycontent>EMPTY</bodycontent>
    <info>torna el valor d'un parametre</info>
    <attribute>
      <name>nomParametre</name>
      <required>true</required>
    </attribute>
  </tag>
  <tag>
    <name>Errors</name>
    <tagclass>formFaces.tags.ListErrorsTag</tagclass>
    <teiclass></teiclass>
    <bodycontent>EMPTY</bodycontent>
    <info>torna la llist a d'errors </info>
    <attribute>
      <name>gravetatMinima</name>
      <required>true</required>
    </attribute>
  </tag>
</taglib>
```

Les quatre primeres serveixen per escriure una llista en una pàgina de quatre maneres diferents: En una taula, en una llista de selecció, en un radio button i en una llista de checkBox.

Les dues següents serveixen per visualitzar un camp d'una classe com un text o com un camp d'entrada.

Les dues següents serveixen per visualitzar un paràmetre de les dues mateixes maneres.

El últim es per llistar els errors que es va guardant el ActionForm, tant de validació com de construcció.

4 - CONSTRUCCIÓ D'UN PROGRAMARI QUE UTILITZI EL FRAMEWORK

4.1 DEFINICIÓ

Es defineix aquest programari d'exemple per utilitzar la potencialitat del Framework, per tant s'ha de veure la flexibilitat i potència de resolució de la pàgina següent en situacions complexes, que la validació de dades d'un formulari al principi evita tornar a carregar el formulari de nou evitant consultes a la base de dades i sobre tot el fàcil i àgil que resulta tant al programador com al dissenyador de pàgines utilitzar-lo.

També intentarem utilitzar el màxim d'etiquetes construïdes per així provar-ne el seu correcte funcionament.

4.2 ANÀLISI

El programari "Captura" que volem construir es una petita aplicació perquè els operaris d'una fàbrica s'assignin a una màquina o línia de producció, seleccionin la feina que faran i la vagin notificant. Es una petita part d'un programari que utilitza connexió directa amb els PLC de les línies de producció per traspassar informació, controla els aturs i els temps de preparació, dóna informació i imatges als operaris dels mètodes, pautes i plànols de les peces, controla els estocs i calcula el rendiment.

En aquesta part els operaris utilitzaran 4 pantalles: la de selecció de màquina, la de selecció de la ordre de treball, la d'identificació del operari amb el seu codi i la que s'utilitza per treballar amb la ordre de treball que ara utilitzarem per notificar la feina feta:

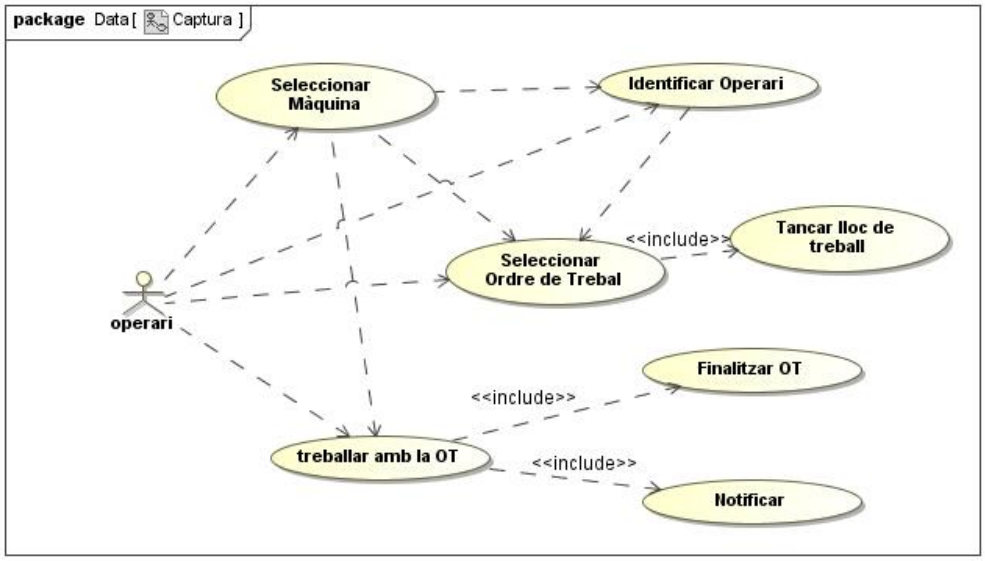


Figura 21: Casos d'us del programa Captura

La web tindrà quatre pantalles i vuit accions, a part tindrà la pantalla Index.jsp que iniciarà la web cridant la acció de consulta de màquines. Les pantalles son:

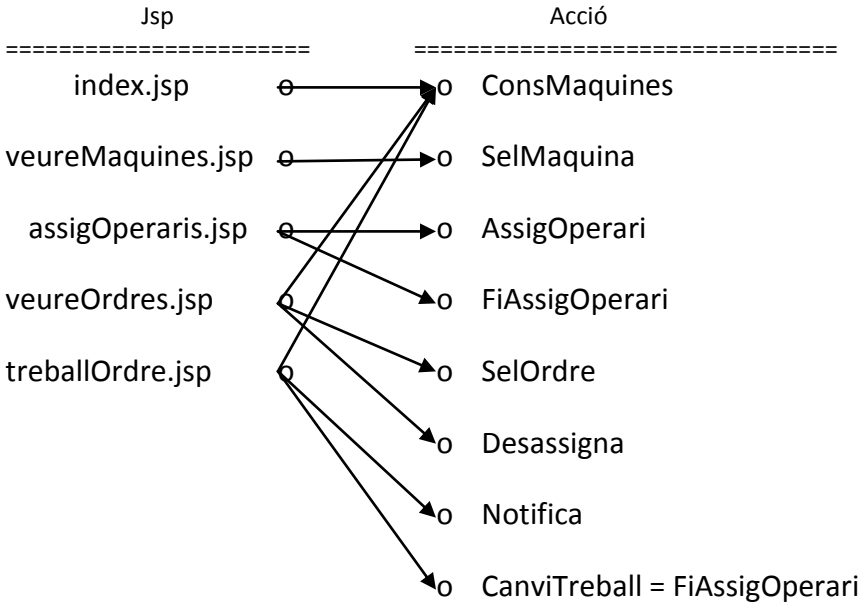
- veureMaquines.jsp Selecció de la màquina a treballar
- assigOperari.jsp Identifica els operaris
- veureOrdres.jsp Selecció de la ordre de treball
- treballOrdre.jsp Treballar amb la ordre

Hi haurà vuit accions que es podran executar des de les diferents pantalles:

- ConsMaquines Troba la llista de les màquines
- SelMaquina S'ha triat una màquina
- AssigOperari S'ha identificat un operari
- FiAssigOperaris S'han acabat d'identificar es operaris
- SelOrdre S'ha seleccionat una ordre
- Desassigna Finalitza la jornada en la màquina
- Notifica S'ha notificat una fabricació
- Canvi Treball Finalitza ordre treball i torna a selecció

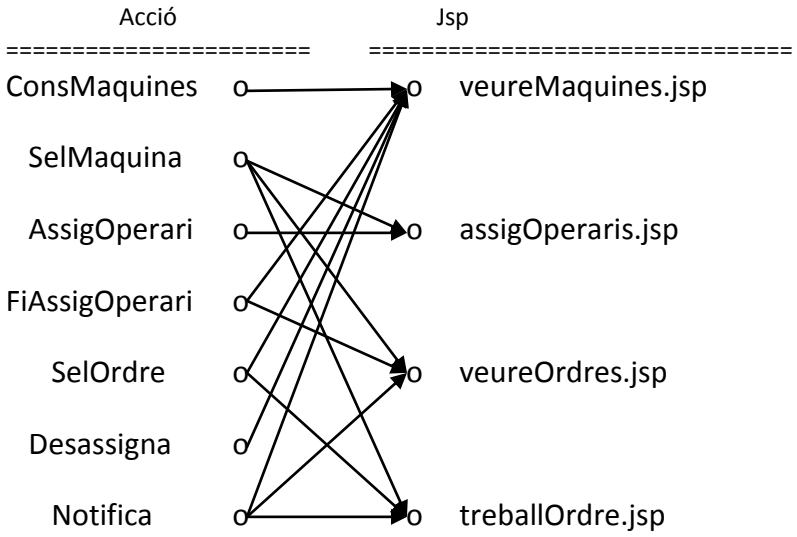
Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

Les accions que es criden en cada pantalla son:



Cada fletxa representa un botó o una selecció i activa una acció. La acció Canvi de treball es com FiassigOperari, per tant la aprofitarem. Aquestes accions inicialitzen la tasca i tornen al formulari veureMaquines o veureOrdres depenent del estat

Les pantalles que pot cridar cada acció son:



Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

Com es veu cada acció pot tornar a diverses pantalles depenent de l'estat en que es troba la màquina. Com des de diferents navegadors es pot accedir al programa es possible que a l'hora d'executar una acció no ens trobem en l'estat apropiat i hem de tenir en compta aquesta possibilitat.

A la pantalla assigOperari.jsp només s'hi ha d'accedir si al voler treballar en una màquina no hi ha ningú assignat.

A la pantalla veureOrdres.jsp només s'hi pot accedir si a la màquina hi ha operaris assignats però no s'està treballant en cap ordre de treball concret

A la pantalla treballOrdre.jsp només s'hi pot accedir si a la màquina hi ha operaris assignats i a més ja tenim una ordre seleccionada.

Per simular una base de dades hem creat la classe GestorDB que manté la base de dades en memòria i actualitza i retorna les dades necessàries per l'aplicació. Així podem fer proves sense haver de tenir cap base de dades concreta.

Les classes que formen part de la base de dades son les següents:

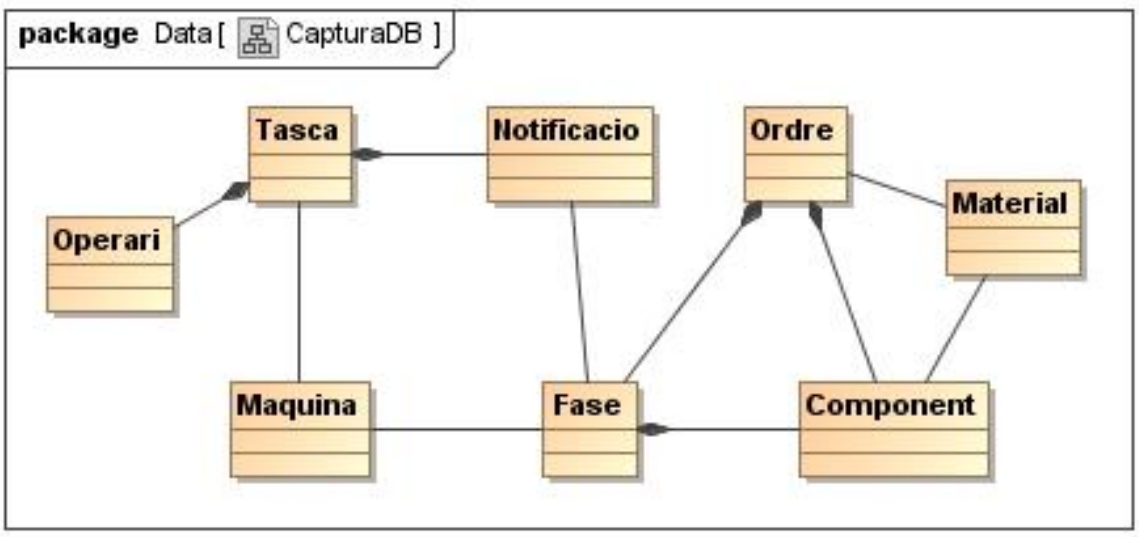


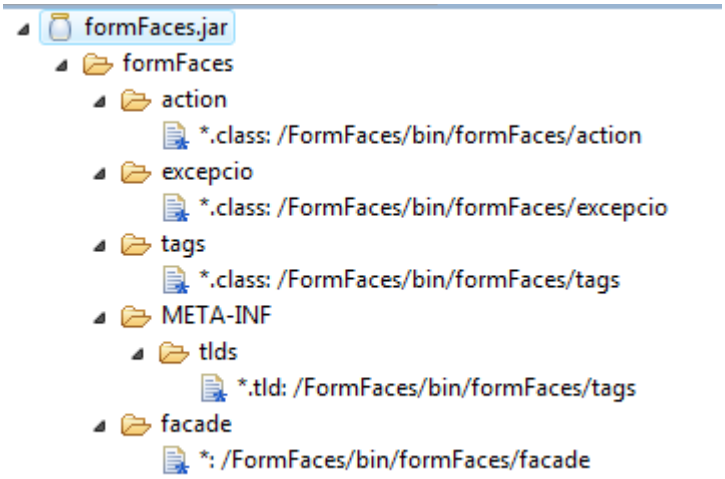
Figura 22: Classes que simulen la base de dades de Captura

Una ordre fabrica un material. Està composada de fases o operacions i de components que formen la llista de materials. Cada fase o operació es fa en una màquina concreta. Cada component es consumeix en una operació en concret.

En les màquines es realitzen tasques, per fer una tasca s'hi assignen un o diversos operaris que realitzen notificacions d'operacions d'ordres.

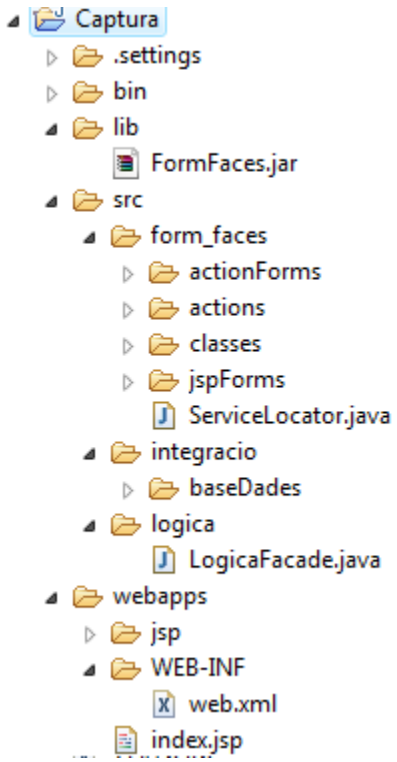
4.3 ENLLAÇ AMB EL NOSTRE FRAMEWORK

Hem construït una llibreria FormFaces.jar amb el següent contingut:



Que té la mateixa estructura que les classes que hem definit al punt 3.3.1 separant el .tld on es defineixen les etiquetes.

Per enllaçar una nova aplicació web amb el nostre framework s'han de seguir els següents passos:



La llibreria FormFaces es col·locarà en la carpeta lib del programa i s'afegirà al classpath.

Es separarà la presentació de la lògica i la integració amb la base de dades, i la presentació es ficarà dins una carpeta form_faces amb subcarpetes: actionForms, actions, classes i jspForms, que amb el seu nom ja indiquen quin contingut i haurem de posar.

Es millor treure el fitxer .tld de la llibreria i ficar-lo dins la carpeta jsp junt amb les pàgines.

Es definirà al fitxer web.xml el servlet que utilitzarem amb el següent contingut:

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Captura</display-name>
  <servlet>
   <servlet-name>ServletFilter</servlet-name>
   <servlet-class>formFaces.facade.ServletFilter</servlet-class>
   <load-on-startup>0</load-on-startup>
  </servlet>

  <!-- Mappings dels servlets -->
  <servlet-mapping>
    <servlet-name>ServletFilter</servlet-name>
    <url-pattern>/ServletFilter/*</url-pattern>
  </servlet-mapping>

  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>

 <!-- The Welcome File List -->
  <welcome-file-list>
   <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

Així queda el servlet que hem construït com a punt de façana del programa. Fet això ja podem començar a construir la nostra aplicació.

4.4 CONSTRUCCIÓ

4.4.1 JSP I JSPFORMS

El jsp **inici.jsp** només farà de punt d'entrada per cridar la primera acció: ConsMaquines que ens mostrarà es màquines per començar triant la màquina amb la que vols treballar:

```
<script>
 function inici() {
   theURL = "/Captura/ServletFilter?action=ConsMaquines";
   document.location=theURL;
   return false;
 }
</script>
</head>
<body onload = "inici()">
</body>
```

Tenim quatre formularis, per tant, també tindrem quatre JspForm un per cada formulari o pantalla. Ara decidirem que volem veure en cada un i els definirem:

veureMaquines: només mostrarà la llista de màquines amb un link per seleccionar-ne una. Per tant la construcció es molt senzilla:

```
import formFaces.action.JspForm;
public class veureMaquinesForm extends JspForm {
   @Override
   protected void definicio() {
      addList("maquines", "Maquina");
   }
}
```

```
<%@ taglib uri="formFaces-taglib.tld" prefix="formfaces" %>
<html><head>
<script>
 function seleccio(maquina) {
   theURL = "/Captura/ServletFilter?action=SelMaquina&maquina="+maquina+"&idForm=veureMaquines";
   document.location=theURL;
   return false;
 }
</script></head>

<body style="background-color:transparent">
   <table >
```


Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

```
        <TR>
            <TD valign="top">
              <formfaces:tableList nomLlista="maquines" camps="codi,descripcio"
              funcio1="selec=seleccio(codi)"/>
            </TD>
        </TR>
      </table>
    </body>
```

Al formulari només cal posar-hi una línia de codi addList per afegir-li la llista maquines.

Al jsp, a part del disseny que se li vulgui donar, només cal posar-hi la etiqueta tableList que llistarà el contingut de la taula maquines, camps codi i descripció i afegirà un link de selecció que cridarà al script selecció passant-li el paràmetre de la llista. Encara que no s'hagi de validar cap camp d'entrada, es millor passar sempre el camp idForm perquè FacesFilter sàpiga quin es.

assigOperaris: només mostrarà la màquina amb la que treballem i la llista d'operaris que ja hem seleccionat. També hi haurà un camp d'entrada per introduir l'operari i la llista d'errors, però això no ha d'estar en el jspForm:

```
public class assigOperarisForm extends JspForm {
  @Override
  protected void definicio() {
    addParametre("maquina", "String");
    addList("operaris", "Operari");
  }
}

<%@ taglib uri="formFaces-taglib.tld" prefix="formfaces" %>
<html><body style="background-color:transparent">
  <table>
   <TR> <TD valign="top">
        <formfaces:tableList nomLlista="operaris" camps="codi,nom"/>
    </TD><TD width="100%" height="80">
      <form action="<%= response.encodeURL("/Captura/ServletFilter?action=AssigOperari" ) %>"
      method="POST">
        <table><tr><td>
          <p align="left"><i><b><font size="4">Màquina:</font></b></i>
        </td><td>
          <formfaces:ParametreInput nomParametre="maquina" tipusInput="HIDDEN"/>
          <input type="HIDDEN" name="idForm" value="assigOperaris">
        </td></tr>
        <tr><td>
          <p align="left"><i><b><font size="4">Operari:</font></b></i>
        </td><td>
          <p align="left"><input type="TEXT" name="operari" size="4">
        </td><td>
          <input type="submit" border="0" value="Assigna">
        </td></tr></table>
      </form>
```


Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

```
    </TD></TR>
    <tr><TD width="100%" height="80">
      <form action="<%= response.encodeURL("/Captura/ServletFilter?action=FiAssigOperari ')
      %>" method="POST">
        <table><tr><td>
          <formfaces:ParametreInput nomParametre="maquina" tipusInput="HIDDEN"/>
          <input type="HIDDEN" name="idForm" value="assigOperaris">
          <input type="submit" border="0" value="Fi Assignacions">
        </td></tr></table>
      </form>
    </TD></TR>
    <tr><td>
      <formfaces:Errors gravetatMinima="0"/>
    </td></tr>
  </table>
</body></html>
```

Al formulari només cal posar-hi el paràmetre maquina per saber sempre en quina màquina treballem i un addList per afegir-li la llista d'operaris ja assignats.

Al jsp, a part del disseny que se li vulgui donar, s'hi ha de posar la etiqueta tableList que llistarà el contingut de la taula operaris, un ParametreInput amb tipus "HIDDEN" per afegir la màquina en tots els forms, un input text per introduir el nou operari a assignar i una etiqueta errors per llistar els errors de validació del operari. També posarem el camp idForm perquè FacesFilter sàpiga quin es. Hi haurà dos formularis, un per assignar el nou operari i un altre per donar per acabada la assignació

veureOrdres: mostrarà la màquina amb la que treballem i la llista d'operaris assignats i la llista d'ordres de treball pendents de la màquina per seleccionar-ne una:

```
public class veureOrdresForm extends JspForm {
  @Override
  protected void definicio() {
    addParametre("maquina", "String");
    addList("operaris", "Operari");
    addList("ordres", "Treball");
  }
}
```

```
<%@ taglib uri="formFaces-taglib.tld" prefix="formfaces" %>
<html>
<head>
<script>
```


Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

```
 function seleccio(ordre,fase) {
  theURL =
"/Captura/ServletFilter?action=SelOrdre&ordre="+ordre+"&fase="+fase+'&idForm=veureOrdres";
  document.location=theURL;
  return false;
 }
</script>
</head>
<body style="background-color:transparent">
  <table width="100%" >
    <TR><TD valign="top">
      <formfaces:tableList nomLlista="operaris" camps="codi,nom"/>
    </TD></TR>
    <TR><TD valign="top">
      <formfaces:tableList nomLlista="ordres" camps="ordre,fase,article,descripcio,pendent"
      funcio1="selec=seleccio(ordre,fase)"/>
    </TD></TR>
      <TR><td><form action="<%= response.encodeURL("/Captura/ServletFilter?action=Desassigna")
      %>" method="POST">
          <table><tr><td>
              <formfaces:ParametreInput nomParametre="maquina" tipusInput="HIDDEN"/>
              <input type="HIDDEN" name="idForm" value="veureOrdres">
          </td><td>
              <input type="submit" border="0" value="Finalizar">
          </td></tr></table>
        </form>
      </TD></TR>
  </table>
</body>
</html>
```

Al formulari només cal posar-hi el paràmetre maquina per saber sempre en quina màquina treballem i les llistes d'operaris ja assignats i d'ordres de treball.

Al jsp, a part del disseny que se li vulgui donar, s'hi ha de posar dues etiquetes tableList una que llistarà el contingut de la taula operaris i una altra que llistarà les ordres de treball amb una funció de selecció. També un ParametreInput amb tipus "HIDDEN" per afegir la màquina en tots els forms. També posarem el camp idForm perquè FacesFilter sàpiga quin es. Es crida a dues accions, una per seleccionar una ordre de treball i l'altre per des assignar els operaris i deixar de treballar en la màquina.

treballOrdre: mostrarà la llista d'operaris assignats, la llista de components de la ordre de treball i les dades del treball (maquina, ordre, fase article i quantitat

pendent). També hi haurà un camp d'entrada per introduir la quantitat a notificar i la llista d'errors, però això no ha d'estar en el jspForm:

```
public class treballOrdreForm extends JspForm {
   @Override
   protected void definicio() {
      addClasse("treball", "Treball");
      addList("operaris", "Operari");
      addList("components", "Component");
   }
}

<%@ taglib uri="formFaces-taglib.tld" prefix="formfaces" %>
<html><body style="background-color:transparent">
   <table width="100%">
    <TR> <TD valign="top"> <formfaces:tableList nomLlista="operaris" camps="codi,nom"/></TD>
     <TD valign="top"> <table>
        <tr><td>Ordre / fase</td>
            <td><formfaces:CampClasse nomClasse="treball" nomCamp="ordre"/></td>
            <td><formfaces:CampClasse nomClasse="treball" nomCamp="fase"/></td>
         </tr><tr><td>Article</td>
            <td><formfaces:CampClasse nomClasse="treball" nomCamp="article"/></td>
            <td><formfaces:CampClasse nomClasse="treball" nomCamp="descripcio"/></td>
        </tr><tr><td>Pendent</td>
            <td><formfaces:CampClasse nomClasse="treball" nomCamp="pendent"/></td>
        </tr><tr><td valign="top">Quantitat</td>
            <td ><form action="<%= response.encodeURL("/Captura/ServletFilter?action=Notifica ") %>"
            method="POST"><table><tr>
            <td><formfaces:CampClasseInput nomClasse="treball" nomCamp="maquina"
            tipusInput="HIDDEN"/>
                <input type="HIDDEN" name="idForm" value="treballOrdre"></td>
            <td><input type="TEXT" name="quantitat" size="4" value="0"></td>
            <td><input type="submit" border="0" value="Notifica"></td>
        </tr></table></form></td>
        </tr><tr><td colspan="3"><formfaces:Errors gravetatMinima="0"/></td></tr></table>
     </TD><TD valign="top" colspan="3">
        <formfaces:tableList nomLlista="components" camps="component,descripcio,qtUnitaria"/>
     </TD><TD width="100%" height="80" colspan="3"><table><tr>
        <td><form action="<%= response.encodeURL("/Captura/ServletFilter?action=FiAssigOperari") %>"
        method="POST"><table><tr><td>
            <formfaces:CampClasseInput nomClasse="treball" nomCamp="maquina" tipusInput="HIDDEN"/>
            <input type="HIDDEN" name="idForm" value="treballOrdre"></td>
            <td><input type="submit" border="0" value="Canviar de treball"></td>
        </tr></table></form></TD>
        <TD><form action="<%= response.encodeURL("/Captura/ServletFilter? action=ConsMaquines ") %>"
        method="POST"><table><tr><td>
            <input type="HIDDEN" name="idForm" value="treballOrdre">
            <input type="submit" border="0" value="Endarrere">
        </td></tr></table></form></TD>
     </tr></table> </TD> </TR></table></body></html>
```

Al formulari només cal posar-hi la classe treball on hi ha totes les dades necessàries de l'ordre de treball i les llistes d'operaris ja assignats i de components de l'ordre de treball.

Al jsp, a part del disseny que se li vulgui donar, s'hi ha de posar dues etiquetes tableList una que llistarà el contingut de la taula operaris i una altra que llistarà els components de l'ordre de treball. També etiquetes campClasse i campClasseInput per mostrar les dades de l'ordre de treball i/o enviar-los a com paràmetre a la nova acció. També posarem el camp idForm perquè FacesFilter sàpiga quin es. Es crida a tres accions en tres formularis diferents, una per notificar peces acabades, una altra (aprofitada FiAssigOperari) per deixar de treballar en l'ordre de treball actual sense sortir de la màquina i anar a seleccionar-ne una altra i la tercera per sortir temporalment del treball sense fer modificacions i tornar al menú principal amb les màquines per si es vol fer una acció des d'aquest terminal en una altra màquina.

4.4.2 ACCIONS I ACTIONFORM

Tenim vuit accions, però dues fan el mateix (CanviTreball i FiAssigOperari) que es inicialitzar el treball actual i mostrar la llista de treballs pendents de la màquina per seleccionar-ne un. Llavors haurem de fer set accions amb els seus set respectius ActionForm. Cada ActionForm ha de contenir els paràmetres d'entrada i les dades que ha de passar als possibles JspForm dels formularis als que pot cridar la acció.

ConsMaquines: Es una acció que es crida al començar sense paràmetres i ha de seleccionar la llista de màquines per passar-les al jsp veureMaquines. Per tant la construcció es molt senzilla:

```
import formFaces.action.ActionForm;

public class ConsMaquinesForm extends ActionForm {
   @Override
   protected void definicio() {
      addList("maquines", "Maquina");
   }
   @Override
   protected void definicioResolucioJsp() {
      addJsp("veureMaquines");
   }
}
```

```
public class ConsMaquines implements Action {
   public void execute(ActionForm af) throws Exception{
      LogicaFacade lf = ServiceLocator.getInstance().getFacade();
      List<Maquines> mqs = lf.getMaquines();
      List<Classe> maqs = new ArrayList<Classe>();
      for (Iterator<Maquines> it = mqs.iterator(); it.hasNext();){
         Maquines mq = it.next();
         Maquina m = new Maquina();
         m.setValor("codi", mq.getCodi());
         m.setValor("descripcio", mq.getDescripcio());
         maqs.add(m);
      }
      af.setLlista("maquines", maqs);
   }
}
```

Al formulari només cal posar-hi la llista de màquines. I com només té un possible formulari següent l'afegim sense condicions.

La acció recupera la llista de màquines de la lògica i la posa al formulari.

SelMaquina: Es una acció complexa que rep el paràmetre màquina i pot tornar tres formularis diferents (assigOperari, veureOrdres o treballOrdre) depenent de l'estat actual de la màquina, per tant haurà de tenir i carregar les dades que hem posat en els tres formularis i resoldre el formulari en qüestió:

```
public class SelMaquinaForm extends ActionForm {
   @Override
   protected void definicio() {
      addParametre("maquina", "String");
      addList("operaris", "Operari");
      addClasse("treball", "Treball");
      addList("components", "Component");
      addList("ordres", "Treball");
   }
   @Override
   protected void definicioResolucioJsp() {
      addJsp("treballOrdre");
      addJspStatus("llista", "operaris");
      addJspStatus("campClasse", "treball.ordre");
      addValidacioStatus("gt", (long)0);
      addJsp("veureOrdres");
      addJspStatus("llista", "operaris");
      addJsp("assigOperaris");
   }
}
```

```
public class SelMaquina implements Action {
   public void execute(ActionForm af) throws Exception{
      LogicaFacade lf = ServiceLocator.getInstance().getFacade();
      Tasques t = lf.getTascaActual(af.getValor("maquina"));
      if (t != null){
         List<String> opes = t.getAssignats();
         List<Classe> ops = new ArrayList<Classe>();
         for (Iterator<String> it = opes.iterator(); it.hasNext();){
            String op = it.next();
            Operari o = new Operari();
            o.setValor("codi", op);
            o.setValor("nom", lf.getNomOperari(op));
            ops.add(o);     }
         af.setLlista("operaris", ops);
         if (t.getOrdreActual()==0){
            List<Ordres> ordres = lf.getOrdres();
            List<Classe> ors = new ArrayList<Classe>();
            for (Iterator<Ordres> it = ordres.iterator(); it.hasNext();){
               Ordres or = it.next();
               for (Iterator<Fases> itf = or.getFases().iterator(); itf.hasNext();){
                  Fases f = itf.next();
                  if (f.getMaquina().equalsIgnoreCase(af.getValor("maquina")) &&
                  or.getPendentFase(f.getNumFase())>0){
                     Treball tr = new Treball();
                     tr.setValor("maquina", af.getValor("maquina"));
                     tr.setValor("ordre", String.valueOf(or.getCodi()));
                     tr.setValor("fase", String.valueOf(f.getNumFase()));
                     tr.setValor("pendent", String.valueOf(or.getPendentFase(f.getNumFase())));
                     tr.setValor("article", or.getArticle());
                     tr.setValor("descripcio", lf.getDescripcioArticle(or.getArticle()));
                     ors.add(tr);
            }  }  }
            af.setLlista("ordres", ors);
         } else {
            Treball tr = new Treball();
            Ordres o = lf.getOrdre(t.getOrdreActual());
            tr.setValor("maquina", af.getValor("maquina"));
            tr.setValor("ordre", String.valueOf(t.getOrdreActual()));
            tr.setValor("fase", String.valueOf(t.getFaseActual()));
            tr.setValor("pendent", String.valueOf(o.getPendentFase(t.getFaseActual())));
            tr.setValor("article", o.getArticle());
            tr.setValor("descripcio", lf.getDescripcioArticle(o.getArticle()));
            af.setClasse("treball", tr);
            List<Components> c = o.getComponents();
            List<Classe> cs = new ArrayList<Classe>();
            for (Iterator<Components> it=c.iterator();it.hasNext();){
               Components co = it.next();
               if (co.getFase()==t.getFaseActual()){
                  Component com = new Component();
                  com.setValor("component", co.getComponent());
                  com.setValor("descripcio", lf.getDescripcioArticle(co.getComponent()));
                  com.setValor("qtUnitaria", String.valueOf(co.getQuantitatUnitaria()));
                  cs.add(com);
            }  }
            af.setLlista("components", cs);
} } } }
```


Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

Al formulari s'hi ha de posar el paràmetre màquina que rebrà, la llista d'operaris que rebran tots tres formularis, la llista d'ordres que rebria el formulari veureOrdres i la llista de components i la Classe treball que rebria el formulari treballOrdre.

com té tres possibles formularis següent els afegim els tres amb addJsp. El primer treballOrdres es resoldrà si la llista d'operaris assignats no està buida i si tenim seleccionat un treball actual (>0). Si no es compleixen aquestes condicions provarà amb el segon veureOrdres que es resoldrà si la llista d'operaris assignats no està buida. Si no es compleix tampoc aquesta condició resoldrà el jsp assigOperaris que no té cap condició.

La acció recupera la tasca actual de la màquines. Si no existeix no fa res i tindrem la llista d'operaris buida. Si existeix carrega la llista d'operaris al formulari i mira si hi ha un treball actual. Si no existeix carrega la llista d'ordres, si existeix carrega el treball actual i la seva llista de components.

AssigOperari: Es una acció per afegir operaris assignats a la màquina. Rep els paràmetres operari i màquina i ha de seleccionar la llista d'operaris assignats per passar-les al jsp assigOperaris. Per tant la construcció es molt senzilla:

```
public class AssigOperariForm extends ActionForm {
   @Override
   protected void definicio() {
      addParametre("maquina", "String");
      addParametre("operari", "String");
      addList("operaris", "Operari");
   }
   @Override
   protected void definicioResolucioJsp() {
      addJsp("assigOperaris");
   }
}
```

```
public class AssigOperari implements Action {
   public void execute(ActionForm af) throws Exception{
      String operari = af.getValor("operari");
      LogicaFacade lf = ServiceLocator.getInstance().getFacade();
      Tasques t = lf.getTascaActual(af.getValor("maquina"));
      if (operari==null || lf.getNomOperari(operari)==null){
         af.novaExcepcio(new FormFacesExcepcio("Operari "+operari+" no existeix",3));
      } else {
         String maq = lf.estaAssignat(operari);
         if (!maq.equalsIgnoreCase("")){
```


Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

```
            af.novaExcepcio(new FormFacesExcepcio("L'operari "+operari+" ja està assignat a la màquina "+maq,3));
        } else {
            if (t == null){
                t = lf.novaTasca(af.getValor("maquina"),operari);
            } else {
                t.addAssignats(operari);
    } } }
    if (t!=null){
        List<String> opes = t.getAssignats();
        List<Classe> ops = new ArrayList<Classe>();
        for (Iterator<String> it = opes.iterator(); it.hasNext();){
            String op = it.next();
            Operari o = new Operari();
            o.setValor("codi", op);
            o.setValor("nom", lf.getNomOperari(op));
            ops.add(o);
        }
        af.setLlista("operaris", ops);
} } }
```

Al formulari només cal posar-hi la llista d'operaris. I com només té un possible formulari següent l'afegim sense condicions.

La acció recupera la tasca actual de la màquines. Valida l'operari, crea la tasca si no existeix, assigna l'operari i recupera la llista d'operaris. En aquesta acció es veu com passar errors a la llista d'errors del formulari per ser tractats per la etiqueta errors.

FiAssigOperari: Quan s'acaben d'assignar tots els operaris. rep el paràmetre màquina i pot tornar dos formularis diferents (veureMaquines i veureOrdres) depenent de si s'han assignat operaris o no. Haurà de tenir i carregar les dades que hem posat en els dos formularis i resoldre el formulari en qüestió:

```
public class FiAssigOperariForm extends ActionForm {
    @Override
    protected void definicio() {
        addParametre("maquina", "String");
        addList("operaris", "Operari");
        addList("ordres", "Treball");
        addList("maquines", "Maquina");
    }
    @Override
    protected void definicioResolucioJsp() {
        addJsp("veureOrdres");
        addJspStatus("llista", "operaris");
        addJsp("veureMaquines");
    }
}
```

```
public class FiAssigOperari implements Action {
   public void execute(ActionForm af) throws Exception{
      LogicaFacade lf = ServiceLocator.getInstance().getFacade();
      Tasques t = lf.getTascaActual(af.getValor("maquina"));
      if (t != null){
         t.setOrdreActual(0);
         t.setFaseActual(0);
         List<String> opes = t.getAssignats();
         List<Classe> ops = new ArrayList<Classe>();
         for (Iterator<String> it = opes.iterator(); it.hasNext();){
            String op = it.next();
            Operari o = new Operari();
            o.setValor("codi", op);
            o.setValor("nom", lf.getNomOperari(op));
            ops.add(o);
         }
         af.setLlista("operaris", ops);
         List<Ordres> ordres = lf.getOrdres();
         List<Classe> ors = new ArrayList<Classe>();
         for (Iterator<Ordres> it = ordres.iterator(); it.hasNext();){
            Ordres or = it.next();
            for (Iterator<Fases> itf = or.getFases().iterator(); itf.hasNext();){
               Fases f = itf.next();
               if (f.getMaquina().equalsIgnoreCase(af.getValor("maquina")) && or.getPendentFase(f.getNumFase())>0){
                  Treball tr = new Treball();
                  tr.setValor("maquina", af.getValor("maquina"));
                  tr.setValor("ordre", String.valueOf(or.getCodi()));
                  tr.setValor("fase", String.valueOf(f.getNumFase()));
                  tr.setValor("pendent", String.valueOf(or.getPendentFase(f.getNumFase())));
                  tr.setValor("article", or.getArticle());
                  tr.setValor("descripcio", lf.getDescripcioArticle(or.getArticle()));
                  ors.add(tr);
         } } }
         af.setLlista("ordres", ors);
      } else {
         List<Maquines> mqs = lf.getMaquines();
         List<Classe> maqs = new ArrayList<Classe>();
         for (Iterator<Maquines> it = mqs.iterator(); it.hasNext();){
            Maquines mq = it.next();
            Maquina m = new Maquina();
            m.setValor("codi", mq.getCodi());
            m.setValor("descripcio", mq.getDescripcio());
            maqs.add(m);
         }
         af.setLlista("maquines", maqs);
} } }
```

Al formulari cal posar-hi el paràmetre màquina, la llista d'operaris i ordres per si es va al jsp veureOrdres i la llista de màquines per si es va al jsp veureMaquines.

com té dos possibles formularis següent els afegim amb addJsp. El primer veureOrdres es resoldrà si la llista d'operaris assignats no està buida. Si no es compleix aquesta condició resoldrà el jsp veureMaquines que no té cap condició.

La acció recupera la tasca actual de la màquina. Si no existeix carrega les màquines i si existeix carrega la llista d'operaris i la d'ordres.

SelOrdre: Rep els paràmetres ordre i fase i pot tornar dos formularis diferents (veureMaquines i treballOrdre) depenent de si troba assignat operaris o no (es poden haver des assignat d'un altre lloc). Haurà de tenir i carregar les dades que hem posat en els dos formularis i resoldre el formulari en qüestió:

```
public class SelOrdreForm extends ActionForm {
   @Override
   protected void definicio() {
      addParametre("ordre", "Long");
      addParametre("fase", "Integer");
      addList("operaris", "Operari");
      addClasse("treball", "Treball");
      addList("components", "Component");
      addList("maquines", "Maquina");
   }
   @Override
   protected void definicioResolucioJsp() {
      addJsp("treballOrdre");
      addJspStatus("llista", "operaris");
      addJspStatus("campClasse", "treball.ordre");
      addValidacioStatus("gt", (long)0);
      addJsp("veureMaquines");
   }
}

public class SelOrdre implements Action {
   public void execute(ActionForm af) throws Exception{
      LogicaFacade lf = ServiceLocator.getInstance().getFacade();
      String maquina = lf.getMaquina(Long.parseLong(af.getValor("ordre")), Integer.parseInt(af.getValor("fase")));
      Tasques t = lf.getTascaActual(maquina);
      if (t != null){
         t.setOrdreActual(Long.parseLong(af.getValor("ordre")));
         t.setFaseActual(Integer.parseInt(af.getValor("fase")));
         List<String> opes = t.getAssignats();
         List<Classe> ops = new ArrayList<Classe>();
         for (Iterator<String> it = opes.iterator(); it.hasNext();){
            String op = it.next();
            Operari o = new Operari();
            o.setValor("codi", op);
            o.setValor("nom", lf.getNomOperari(op));
            ops.add(o);
         }
```

```
            af.setLlista("operaris", ops);
            Treball tr = new Treball();
            Ordres o = lf.getOrdre(t.getOrdreActual());
            tr.setValor("maquina", maquina);
            tr.setValor("ordre", String.valueOf(t.getOrdreActual()));
            tr.setValor("fase", String.valueOf(t.getFaseActual()));
            tr.setValor("pendent", String.valueOf(o.getPendentFase(t.getFaseActual())));
            tr.setValor("article", o.getArticle());
            tr.setValor("descripcio", lf.getDescripcioArticle(o.getArticle()));
            af.setClasse("treball", tr);
            List<Components> c = o.getComponents();
            List<Classe> cs = new ArrayList<Classe>();
            for (Iterator<Components> it=c.iterator();it.hasNext();){
               Components co = it.next();
               if (co.getFase()==t.getFaseActual()){
                  Component com = new Component();
                  com.setValor("component", co.getComponent());
                  com.setValor("descripcio", lf.getDescripcioArticle(co.getComponent()));
                  com.setValor("qtUnitaria", String.valueOf(co.getQuantitatUnitaria()));
                  cs.add(com);
            } }
            af.setLlista("components", cs);
         } else {
            List<Maquines> mqs = lf.getMaquines();
            List<Classe> maqs = new ArrayList<Classe>();
            for (Iterator<Maquines> it = mqs.iterator(); it.hasNext();){
               Maquines mq = it.next();
               Maquina m = new Maquina();
               m.setValor("codi", mq.getCodi());
               m.setValor("descripcio", mq.getDescripcio());
               maqs.add(m);
            }
            af.setLlista("maquines", maqs);
} } }
```

Desassigna: Acaba de treballar en una màquina i torna a mostrar la llista de màquines. Rep el paràmetre màquina i torna el formulari veureMaquines. Haurà de carregar les màquines:

```
public class DesassignaForm extends ActionForm {
   @Override
   protected void definicio() {
      addParametre("maquina", "String");
      addList("maquines", "Maquina");
   }
   @Override
   protected void definicioResolucioJsp() {
      addJsp("veureMaquines");
   }
}
```

```
public class Desassigna implements Action {
   public void execute(ActionForm af) throws Exception{
      LogicaFacade lf = ServiceLocator.getInstance().getFacade();
      lf.desassigna(af.getValor("maquina"));
      List<Maquines> mqs = lf.getMaquines();
      List<Classe> maqs = new ArrayList<Classe>();
      for (Iterator<Maquines> it = mqs.iterator(); it.hasNext();){
         Maquines mq = it.next();
         Maquina m = new Maquina();
         m.setValor("codi", mq.getCodi());
         m.setValor("descripcio", mq.getDescripcio());
         maqs.add(m);
      }
      af.setLlista("maquines", maqs);
} }
```

Notifica: notifica un treball. Rep els paràmetres màquina i quantitat i valida que la quantitat sigui més gran que zero, si notifica tota la quantitat pendent de la fase, la finalitza i se n'haurà de seleccionar una altra. Si la quantitat no es positiva es tornarà el jsp guardat pel FacesFilter sense haver de cridar la lògica de la aplicació. Pot tornar als jsp veureMaquines, veureOrdres o treballOrdre pel que el formulari haurà de tenir les dades dels tres JspForm.

```
public class NotificaForm extends ActionForm {
   @Override
   protected void definicio() {
      addParametre("maquina", "String");
      addParametre("quantitat", "Float");
      addValidacio("gt", (float)0);
      addList("operaris", "Operari");
      addClasse("treball", "Treball");
      addList("components", "Component");
      addList("ordres", "Treball");
      addList("maquines", "Maquina");
   }
   @Override
   protected void definicioResolucioJsp() {
      addJsp("treballOrdre");
      addJspStatus("llista", "operaris");
      addJspStatus("campClasse", "treball.ordre");
      addValidacioStatus("gt", (long)0);
      addJsp("veureOrdres");
      addJspStatus("llista", "operaris");
      addJsp("veureMaquines");
   }
}
```

```
public class Notifica implements Action {
   public void execute(ActionForm af) throws Exception{
      LogicaFacade lf = ServiceLocator.getInstance().getFacade();
      Tasques t = lf.getTascaActual(af.getValor("maquina"));
      if (t != null){
         if (t.getOrdreActual()>0){ lf.notifica(t, Float.parseFloat(af.getValor("quantitat")));}
         List<String> opes = t.getAssignats();
         List<Classe> ops = new ArrayList<Classe>();
         for (Iterator<String> it = opes.iterator(); it.hasNext();){
            String op = it.next();
            Operari o = new Operari();
            o.setValor("codi", op);
            o.setValor("nom", lf.getNomOperari(op));
            ops.add(o);
         }
         af.setLlista("operaris", ops);
         if (t.getOrdreActual()!=0){
            Treball tr = new Treball();
            Ordres o = lf.getOrdre(t.getOrdreActual());
            tr.setValor("maquina", af.getValor("maquina"));
            tr.setValor("ordre", String.valueOf(t.getOrdreActual()));
            tr.setValor("fase", String.valueOf(t.getFaseActual()));
            tr.setValor("pendent", String.valueOf(o.getPendentFase(t.getFaseActual())));
            tr.setValor("article", o.getArticle());
            tr.setValor("descripcio", lf.getDescripcioArticle(o.getArticle()));
            af.setClasse("treball", tr);
            List<Components> c = o.getComponents();
            List<Classe> cs = new ArrayList<Classe>();
            for (Iterator<Components> it=c.iterator();it.hasNext();){
               Components co = it.next();
               if (co.getFase()==t.getFaseActual()){
                  Component com = new Component();
                  com.setValor("component", co.getComponent());
                  com.setValor("descripcio", lf.getDescripcioArticle(co.getComponent()));
                  com.setValor("qtUnitaria", String.valueOf(co.getQuantitatUnitaria()));
                  cs.add(com);
            } }
            af.setLlista("components", cs);
         } else {
            List<Ordres> ordres = lf.getOrdres();
            List<Classe> ors = new ArrayList<Classe>();
            for (Iterator<Ordres> it = ordres.iterator(); it.hasNext();){
               Ordres or = it.next();
               for (Iterator<Fases> itf = or.getFases().iterator(); itf.hasNext();){
                  Fases f = itf.next();
                  if (f.getMaquina().equalsIgnoreCase(af.getValor("maquina")) && or.getPendentFase(f.getNumFase())>0){
                     Treball tr = new Treball();
                     tr.setValor("maquina", af.getValor("maquina"));
                     tr.setValor("ordre", String.valueOf(or.getCodi()));
                     tr.setValor("fase", String.valueOf(f.getNumFase()));
                     tr.setValor("pendent", String.valueOf(or.getPendentFase(f.getNumFase())));
                     tr.setValor("article", or.getArticle());
                     tr.setValor("descripcio", lf.getDescripcioArticle(or.getArticle()));
                     ors.add(tr);
            } } }
```

```
        af.setLlista("ordres", ors);
      }
    } else {
      List<Maquines> mqs = lf.getMaquines();
      List<Classe> maqs = new ArrayList<Classe>();
      for (Iterator<Maquines> it = mqs.iterator(); it.hasNext();){
        Maquines mq = it.next();
        Maquina m = new Maquina();
        m.setValor("codi", mq.getCodi());
        m.setValor("descripcio", mq.getDescripcio());
        maqs.add(m);
      }
      af.setLlista("maquines", maqs);
} } }
```

L'interessant d'aquest formulari es comprovar com funciona el facesFilter i la validació del ActionForm si la quantitat no es positiva o s'introdueix un caràcter no numèric a la casella de la quantitat: El facesFilter recupera el jsp anterior i si la validació dels paràmetres d'entrada de l'ActionForm no es correcta no s'executa cap acció i carrega els errors de validació. Al trobar el filterChain que la aplicació no retorna cap formulari, accepta com a bo el que ha carregat el FacesFilter.

4.2.3 CLASSES

Per fer aquesta web hem necessitat definir aquestes quatre classes:

```
public class Component extends Classe {
  @Override
  protected void definicio() {
    addParametre("component", "String");
    addParametre("descripcio", "String");
    addParametre("qtUnitaria", "Float");
  }
}

public class Maquina extends Classe {
  @Override
  protected void definicio() {
    addParametre("codi", "String");
    addParametre("descripcio", "String");
  }
}
```

```
public class Operari extends Classe {
  @Override
  protected void definicio() {
    addParametre("codi", "String");
    addParametre("nom", "String");
  }
}

public class Treball extends Classe {
  @Override
  protected void definicio() {
    addParametre("maquina", "String");
    addParametre("ordre", "Long");
    addParametre("fase", "Integer");
    addParametre("pendent", "Float");
    addParametre("article", "String");
    addParametre("descripcio", "String");
  }
}
```

4.5 PROVES

S'han fet proves de totes les accions provocant errors de validació, entrant amb dues sessions del navegador al mateix temps, tancant el navegador i tornant a entrar en diferents situacions, posant caràcters alfabètics en comptes de numèrics i el Framework ha treballat perfectament.

5 - MILLORES A FER EN EL FRAMEWORK

Durant el quadrimestre que dura el projecte es pot fer un primer esborrany de Framework però queden moltes millores a fer que no dóna temps d'implementar. Enumerarem algunes millores que hem anat pensant durant la construcció i les proves i que posarem en pràctica en un futur proper:

- Tractament de classes: El framework ha de poder tractar les classes normals que s'utilitzen a la lògica de negoci. Una forma senzilla d'implementar-ho es construint un map a la classe abstracta Classe que reculli els paràmetres i les llistes de les classes de la lògica.
- Filtres: S'ha d'acabar el sistema de carregar possibles filtres des d'un XML. Es una funcionalitat molt senzilla: en un filter.xml es definirà <filter>classe</filter> i el constructor del ServletFilter els carregarà. La funcionalitat perquè passin al ChainFilter ja funciona.
- Ampliar els tipus de camp en les classes.
- Donar més potencialitat al FacesFitre perquè passi al jsp els paràmetres que passen en el request, així no s'hauran de tornar a introduir els paràmetres bons.
- Millorar les etiquetes: posar atributs per millorar la presentació com: colorColumna="String",String", ……,"String" o color="String" o ampleColumna= int, int, ……, int. O alturaFiles etc.

Autor: Jaume Martí Muixí
Consultor: Josep Maria Camps Riba

6 - CONCLUSIONS

En aquest projecte s'ha vist com es pot construir un entorn per facilitar la feina dels desenvolupadors d'aplicacions. S'ha vist amb quina facilitat es pot construir una aplicació web utilitzant el Framework, a més, es construeix seguint un bon disseny.

Hem estudiat els millors exemples de Framework de presentació que hi ha al mercat: Struts, Struts2, Spring i JSF. De tots n'hem tret coses interessants, d'entre elles cal citar:
De Struts 2 n'hem tret el concepte dels filtres que es molt potent per donar funcionalitats generals a totes les peticions.
De Struts n'hem obtingut el concepte de les accions i dels formularis que faciliten molt el disseny del patró MVC.
De JSF n'hem tret el concepte de la validació dels paràmetres d'entrada i tornar el jsp sense haver de cridar la lògica per reconstruir-lo.
Hem aprofitat la potencia del disseny d'etiquetes pròpies per encapsular les classes del framework i facilitar la feina als dissenyadors.

Mentre l'hem dissenyat hem estudiat patrons de disseny, gràcies a ells hem aconseguit una arquitectura ben dissenyada i de fàcil manteniment. L'estudi dels patrons ens ha proporcionat maneres elegants de resoldre problemes complexos amb els que ens hem trobat.

Ha estat un treball realment interessant que anirem millorant i espero sigui de mota utilitat.

7 BIBLIOGRAFIA

http://en.wikipedia.org/wiki/Apache_Struts

http://static.springsource.org/spring/docs/current/spring-framework-reference/html/

http://download.oracle.com/javaee/5/tutorial/doc

http://java.sun.com/developer/Books/javaserverpages/cservletsjsp/chapter14.pdf

8 FIGURES