TEMARIO OPOSICIÓNS INFORMÁTICA

GRUPO A1 - ESCALA DE SISTEMAS E TECNOLOXÍA DA INFORMACIÓN

TEMA 31. PATRÓNS DE DESEÑO E FRAMEWORKS. MVC. JSF. ANTIPATRÓNS.

Autor: Juan Marcos Filgueira Gomis

TEMA 31. PATRÓNS DE DESEÑO E FRAMEWORKS. MVC. JSF. ANTIPATRÓNS.

30.1 INTRODUCIÓN E CONCEPTOS

Para moitos problemas de deseño que se repiten tanto en desenvolvementos software como en implantacións hardware existen solucións comúns de aplicación dentro do mesmo contexto. Estas solucións recorrentes denomínanse **patróns de deseño** e se basean no concepto de reutilización e aproveitamento de solucións xa existentes en problemas novos. Os patróns segundo o autor acostuman a dividirse en diferentes familias, sendo a clasificación máis habitual en patróns de deseño, de arquitectura e interacción. Á súa vez dentro dos patróns de deseño clasifícanse segundo creacionais, estruturais e de comportamento. Así mesmo defínense os patróns de programación como patróns específicos para linguaxes de programación ou sistemas concretos. Cómpre sinalar que nunha mesma solución ou deseño pode convivir calquera número de patróns que sexa necesario, xa que en moitos casos trátase de solucións parciais a problemas concretos non de solucións xerais. Fronte ao concepto de patrón xorde o de **antipatrón** que definen erros de deseño comúns ou problemas que se repiten a miúdo para axudar a identificalos. Os patróns e antipatróns poden definirse perante linguaxes de definición do estilo da Linguaxe Unificada de Modelado ou **UML** (en inglés *Unified Modeling Language*), que soportado polo OMG é un dos máis empregados actualmente.

Nas aplicacións JEE e .NET un dos patróns de uso máis estendido é o Modelo Vista Controlador ou **MVC** (en inglés *Model View Controller*) que se basea na separación dunha aplicación en tres capas ou compoñentes diferenciados, interface de usuario, lóxica de negocio e sistemas de información.

Este patrón intégrase en ***frameworks***, ou compoñentes software que implementan funcionalidades comúns a conxuntos de aplicacións, e que poden seguir os modelos de patróns de deseño. O patrón sería a solución de deseño abstracta e o *framework* unha implementación do mesmo concreta. Algúns *frameworks* como Struts representan o esqueleto dunha aplicación, con implementación do patrón MVC entre outros, aportando así directamente todas as funcionalidades precisas para o seu funcionamento interno. Se nunha mesma aplicación engadimos novos *frameworks* disporemos de funcionalidades engadidas. Deste xeito o *framework* **JSF** (en inglés *Java Server Faces*) proporciona, complementariamente a Struts ferramentas que facilitan o desenvolvemento de interfaces de usuario.

31.2 PATRÓNS DE DESEÑO E FRAMEWORKS

As principais vantaxes do emprego de patróns en solucións software pasan por facilitar a comunicación interna entre compoñentes, aforrar tempo e outros recursos, mellorar a calidade das operacións de todo o ciclo de vida de desenvolvemento e facilitar a aprendizaxe. A día de hoxe é un feito consumado que a súa aplicación correcta reporta un beneficio directo en calquera desenvolvemento ou implantación.

31.2.1 Clasificación xeral

Existen moitas clasificacións dos patróns, segundo o autor(es) pero a máis habitual fai referencia ao ámbito de aplicación do patrón tomando como referencia a enxeñaría do software:

1) **Patróns de deseño.** Proporcionan un esquema de aplicación en partes dun sistema software. Definen estruturas que resolven un problema de deseño de utilidade en diferentes aplicacións.
2) **Patróns de arquitectura.** Proporcionan un esquema ou organización estrutural para definir sistemas completos ou subsistemas incluíndo responsabilidades e relacións entre sistemas.
3) **Patróns de interacción.** Proporcionan un deseño de interface para aplicacións ou aplicacións web.
4) **Patróns de programación** (en inglés *Idioms patterns*). Patróns a baixo nivel para linguaxes de programación ou tecnoloxías específicas. Definen representacións de implementacións de compoñentes e relacións considerando funcionalidades propias de cada linguaxe.

31.2.2 Clasificación de patróns para tecnoloxías de servidores de aplicacións

A maiores foron xurdindo patróns para outros ámbitos de aplicación, como programación multifío, fluxos de traballo para procesos de sistemas empresariais, arquitecturas SOA ou integración de sistemas. En definitiva, pode concluírse que o concepto de patrón pode estenderse a calquera problema que nos atopemos e o nivel de abstracción que precisemos na solución. Existen diferentes catálogos de patróns, sendo os máis coñecidos:

- ✔ **GoF**, (en inglés *Gang of Four*) para problemas de deseño. (1995).
- ✔ **POSA**, (en inglés *Pattern Oriented Software Architecture*) para solucións en arquitecturas SOA. (1996).
- ✔ **J2EE**, para solucións específicas desta tecnoloxía. (2003).
- ✔ **PoEAA** (en inglés *Patterns of Enterprise Application Architecture*). Para sistemas complexos en arquitecturas empresariais distribuídas en capas. (2003).
- ✔ **GRASP** (en inglés *General Responsibility Assignment Software Patterns*). Patróns xerais para asignación de responsabilidades e transicións. (2005).

En concreto para o ámbito dos servidores de aplicacións como .NET e JEE en canto a arquitectura e análise poden destacarse os seguintes atendendo ao seu nivel de utilización:

a) **Patrón de análise Party (Grupo).** Agrupa as responsabilidades similares dos tipos de colectivos dunha organización nun supertipo. Emprégase para facilitar o modelado de estruturas en organización, sendo cada tipo unha organización, empresa, rol ou papel e almacenar os datos persoais de cada membro. Situacións especiais obrigan a adaptacións deste patrón como ocorre no Party Type Generalizations que permite a xeneralización de tipos de grupo que herdan dun subtipo, por exemplo para unha persoa ten varios roles a un tempo.

b) **Patrón de análise Accountability.** Establece unha relación de responsabilidade entre dúas partes ou perfís. Cos tipos Accountability e Accountability Type permite expresar a clase de relación entre ambos. Pode facer uso do patrón Party para obter unha maior flexibilidade. Segundo sexan as relacións pode dar lugar a patróns máis complexos como Hierarchic Accountability ou Xerarquía de responsabilidade que engade restricións aos elementos de responsabilidade; ou que ten aplicación á hora de delegar tipos de responsabilidade a un subpatrón Party.

c) **Patrón arquitectónico MVC** (en inglés *Model View Controller*). Estrutura un compoñente software en 3 capas, o modelo coa lóxica de negocio, funcionalidades e sistemas de información, a vista coa interface de usuario e o Controlador que recibe os eventos da entrada e coordina as actividades da vista.

d) **Patrón arquitectónico PAC** (en inglés *Presentation Abstraction Control*). Similar ao MVC este patrón define un sistema interactivo baseado nunha xerarquía de axentes cooperantes que realizan funcionalidades concretas. Divídese en tres capas: Presentación con interacción persoa-máquina, Abstracción coa lóxica e sistemas de información e o Control que centraliza as comunicacións entre axentes, procesa eventos externos e actualiza o modelo. A principal diferenza co MVC radica en que se poden facer diferentes axentes ou subsistemas de aplicación, operando de forma independente ou xerarquizada.

e) Patrón arquitectónico Capas (en inglés *Layers*). Representaría a abstracción xenérica dos patróns anteriores a un sistema multicapa, orientado cara a distribución xerárquica de roles e responsabilidades. Permite aumentar ou diminuír o nivel de abstracción, máis ou menos capas, illando o mantemento e actualización de cada capa. Cada nivel ou capa ofrece servizos á capa superior e usa os da inferior.

f) Patrón arquitectónico Pipes and Filters. Orientado tamén a arquitecturas SOA, neste modelo cada compoñente posúe un conxunto de entradas e saídas. Representa a lectura de fluxos de datos, transformándoos nun fluxo de saída sen ter que procesar toda a entrada, como ocorre nos modelos Streaming e de aí que se denominen Filtros aos compoñentes que reciben as entradas e tuberías ou condutos aos que encamiñan o fluxo cara a saída. Permite representar procesamentos en paralelo así coma execución concurrente.

g) Patrón arquitectónico Blackboard. Proporciona un modelo de solucións aproximadas, cando non se pode aplicar unha solución concreta. Permite reutilizar as fontes de coñecemento e un mellor soporte de cambios e mantemento da solución aproximada.

h) Patrón Microkernel. Dentro dos patróns para sistemas adaptables, este modelo separa un kernel funcional mínimo do estendido para soportar sistemas software con requirimentos que cambian ao longo do tempo. Ideado para sistemas operativos, cada un deles sería unha vista do Microkernel central, permitindo que se poida estender o sistema de xeito doado.

i) Patrón Reflection. Outro patrón sistemas adaptables que modela un mecanismo para mudar a estrutura e comportamento dun sistema dinamicamente. Establece dous niveis: Metadatos para que os software leve unha descrición de si mesmo e Lóxica de aplicación. Os cambios de comportamento poden reflectirse nos metadatos, pero isto pode pasar inadvertido.

j) **Patrón arquitectónico Broker.** Orientado a arquitecturas SOA e sistemas distribuídos onde varios clientes fan peticións a un servidor ou servizo remoto. O axente Broker encárgase de coordinar a comunicación entre o cliente e o provedor do servizo. As principais vantaxes deste patrón son permitir a transparencia de localización do servizo, permitir cambios e ampliación de novos compoñentes sen que o sistema se vexa afectado, mellora da portabilidade e interoperabilidade con outros axentes Broker.

k) **Patrón Publisher Subscriber.** Orientado a arquitecturas SOA e sistemas distribuídos insire unha capa entre clientes e servidores que se encarga de levar conta da comunicación de xeito transparente. Representa unha arquitectura de mensaxería sen acoplamento.

No tocante ao deseño, os principais patróns acostuman a agruparse en tres grandes categorías: Patróns creacionais, estruturais e de comportamento. Os **creacionais** incluirían:

a) **Abstract Factory.** Prove unha interface que permite a creación de familias de obxectos dependentes ou relacionas sen ter que especificar as clases completas. Exemplos deste patrón serían os Widgets e compoñentes de interfaces gráficas.

b) **Builder**. Construtor virtual que separa a construción dun obxecto complexo da súa representación, de tal xeito que se obteñen diferentes representación nun mesmo proceso.

c) **Factory Method.** Patrón que define unha interface para a creación de obxectos deixando que as subclases decidan que clase instanciar, facendo que o proceso de xeración do subtipo sexa transparente ao usuario.

d) **Prototype**. Permite a creación de novos obxectos clonándoos dunha instancia dun obxecto xa existente.

e) **Singleton**. Patrón de instancia única que asegura que dunha clase só existirá unha única instancia definindo un punto de acceso común á mesma.

f) **Object Pool.** Patrón para a obtención de obxectos por clonación. Crease unha instancia dun tipo de obxecto da clase a clonar. Está pensado para casos onde a creación teña un custe moi alto e se permita a utilización de obxectos xenéricos do Pool.

Por outra banda, dentro dos **estruturais**:

a) **Adapter**. Patrón que convirte a interface dunha clase noutra interface adaptada a necesidades específicas como determinados clientes ou interfaces requiridas por compatibilidade.

b) **Bridge**. Ou patrón Handle/Body, separa unha abstracción da súa implementación de xeito que ámbalas dúas podan mudar de forma de maneira independente, sen que cambios nunha afecten a outra.

c) **Composite**. Patrón que permite manipular obxectos compostos coma se de un simple se tratase. Fai uso da composición recursiva e a estruturas en forma de árbore para poder presentar unha interface común.

d) **Decorator**. Responde á necesidade de engadir funcionalidades a obxectos dinamicamente. Crea unha xerarquía de clases onde as fillas herdan da nais as funcionalidades e incorporan as súas propias.

e) **Facade**. Proporciona unha interface común de acceso a un conxunto de interfaces dun sistema. Facilita o emprego do sistema interno con outras interfaces de alto nivel. Os clientes só poden comunicarse a través da interface única que fai de fachada.

f) **Flyweight**. Permite eliminar a redundancia entre obxectos que presentan a mesma información. Factoriza os atributos comúns a estes obxectos nunha clase lixeira.

g) **Proxy**. Proporciona un punto de control de acceso ou intermediario para o control doutro(s) obxecto(s). Presenta diferentes niveis de aplicabilidade:

   - ✔ *Proxy remoto.* Representa a un obxecto remoto de xeito local, codificando a petición e argumentos antes de enviala ao obxecto remoto.
   - ✔ *Proxy virtual.* Crea obxectos de alto custe baixo demanda, con posibilidade de caché da información dos mesmos limitando os custes de acceso.
   - ✔ *Proxy de protección*. Controla o acceso a obxectos remotos comprobando que os clientes dispoñen dos permisos necesarios.
   - ✔ *Proxy de referencia intelixente*. Análogo a un punteiro con operacións adicionais sobre un obxecto para temas de concorrencia, acceso a memoria, etc...

O último bloque serían os patróns de comportamento:

a) **Chain of responsibility.** O patrón cadea de responsabilidade permite establecer a liña que deben levar as mensaxes, denominada cadea de obxectos receptores, permitindo que varios obxectos podan capturar unha mensaxe, como pode ser unha excepción Java. Calquera dos receptores podería responder á petición segundo o criterio establecido.

b) **Comando ou Orde.** Patrón que encapsula unha operación nun obxecto, de xeito que se poidan facer operacións estendidas como almacenamento e colas de peticións e soporte de accións de facer e desfacer.

c) **Intérprete**. Define unha representación para a gramática dunha linguaxe xunto co seu intérprete.

d) **Iterator**. Patrón co obxectivo de permitir percorrer obxectos compostos como poden ser as coleccións sen necesidade de contemplar aspectos de implementación ou representación interna dos mesmos. Define unha interface onde se ofrecen diferentes métodos para percorrer o obxecto complexo.

e) **Mediador**. Define un obxecto que facilita a interacción entre outros de distinto tipo, coordinando a comunicación entre eles. O obxectivo é encapsular a interacción deses obxectos para evitar o acoplamento entre eles.

f) **Memento**. Representa o estado dun obxecto ou sistema complexo para permitir o seu almacenamento e modificación, de xeito que se poida restaurar volvendo a estados anteriores no tempo.

g) **Observador**. Permite definir unha dependencia dun a moitos, de xeito que eventos ou modificacións de estado disparen a notificación dos cambios a todos os obxectos ou sistemas dependentes.

h) **Estado.** Emprégase para permitir que un obxecto cambie o seu comportamento no caso de modificarse o seu estado. Deste xeito diferentes clases poden representar a un mesmo obxecto ao longo do tempo.

i) **Estratexia**. Permite definir unha familia de algoritmos ou métodos de resolución, permitindo seleccionar dinamicamente cales aplicar e que deste xeito sexan intercambiables.

j) **Template Method.** Define o esqueleto dun algoritmo para unha operación, delegando partes do mesmo ás clases concretas. Deste xeito as subclases poden redefinir pasos concretos do método de resolución.

k) **Visitor**. Representa un algoritmo ou operación realizada sobre a estrutura dun obxecto, permitindo a definición de novas operacións sen altera o tipo dos elementos sobre os que se realiza a operación.

Nunha última categoría poderían incluírse os patróns propias de linguaxes de programación ou tecnoloxías concretas, sendo os **patróns JEE**, a maioría Core J2EE Patterns, aqueles cun uso máis estendido dentro do mundo dos servidores de aplicacións e os servizos web:

a) **Intercepting Filter.** Intercepta as peticións da capa de presentación antes ou despois do seu procesamento permitindo realizar operacións sobre os datos como auditorías, comprobacións de seguridade, conversións ou validacións. Permiten conectarse en fervenza e activar ou desactivar sen que afecte ao funcionamento xeral dunha aplicación. Permite diferentes estratexias como: Custom Filter, Estándar, Base Filter e Template Filter.
b) **Front Controller.** Centraliza o control das peticións da capa de presentación, dirixíndoas cara o compoñente axeitado para validación de parámetros, invocación de elementos da lóxica de negocio, etc... Un controlador encárgase de recoller as peticións e factorizar o código repetitivo.
c) **View Helper**. Prove unha clase que engloba código común, con aplicación tanto para a capa de negocio como para a de presentación. Cada vista contén código para formato, delegando as responsabilidades de procesamento nas clases de axuda implementadas como Java Beans ou Custom Tags. Así mesmo poden almacenar modelos de datos intermedios facendo adaptacións previas do negocio, como conversións ou validacións, o lóxico pola separación en capas é que estas operacións non sexan moi complexas.
d) **Composite View**. Define unha xerarquía de vistas compostas de diferentes vistas particulares permitindo modificar as partes en tempo de execución e a partir de modelos. Deste xeito inclúense dinamicamente as vistas concretas en vistas compostas da aplicación a través dos mecanismos que dispoñen para tal efecto JSP e Servlets.
e) **Service to worker**. Agrupa varios patróns a modo de *framework* para permitir combinar un controlador (Front Controller), e un Dispatcher ou controlador de vistas (View Helper), para manexar as peticións dos clientes e xerar a presentación dinamicamente como resposta. Os controladores solicitan o contido aos Helpers que enchen o modelo de negocio intermedio.
f) **Dispatcher View**. Cunha estrutura similar á do Service to worker, neste modelo tanto Controlador como Dispatcher teñen responsabilidades máis limitadas xa que lóxica de procesamento e control da vista son básicas.
g) **Business Delegate**. Permite a abstracción a implementación de compoñentes complexos como EJB ou JMS da capa de presentación. Deste xeito poden crearse clases Proxy que almacenen e encolen as peticións podendo proporcionar control de prioridades, xestión de excepcións ou caché. O patrón emprega un compoñentes denominado Lookup Service, responsable de ocultar os detalles de implementación do código de busca dentro da lóxica de negocio.

h) **Value Object (VO)**. Encapsula un conxunto de datos que representan un obxecto ou entidade do negocio. Cando se solicita a un Bean un conxunto de información este pode crear o obxecto Value Object e encher os seus atributos para devolvelo ao cliente.

i) **Session Facade**. Emprega un Bean de sesión como fachada para encapsular as interaccións dos compoñentes de negocio e ofrecer un servizo de acceso uniforme, a través dos interfaces requiridos unicamente a través dos casos de uso. Proporciona unha abstracción de alto nivel implementada a modo de Bean.

j) **Composite Entity.** Permite ampliar os Beans de entidade cando estes son de pequeno tamaño, deste xeito poden aumentarse mantendo a compatibilidade. O abuso deste patrón considérase un antipatrón xa que pode dar lugar a estruturas moi complexas. Un Bean Composite Entity representa un grafo de obxectos, por tanto debe empregarse con coidado.

k) **Transfer Object Assembler**. Simplifica o acceso aos sistemas de información a través dun conectador común. Cada obxecto de negocio terá un Transfer Object (TO) cos detalles de acceso a datos (Beans, JDO, JDBC, ...) e un Bean de sesión funcionará como interface común.

l) **Service Locator**. Emprégase para abstraer a utilización de JNDI a través dun obxecto Service Locutor e para ocultar as complexidades da creación do contexto inicial, así como da busca e instanciación de EJBs a través dun punto de acceso común.

m) **Data Access Object (DAO)**. Emprégase un obxecto como medio de acceso a sistemas de información, en especial Bases de datos. Abstrae e encapsula as operacións relacionadas coa tecnoloxía de persistencia empregada (JDBC, JDO, LDAP, Beans, TopLink, Hibernate, iBATIS, etc...). Controla os parámetros de conexión, obtención de datos e almacenamento proporcionando unha interface de acceso común.

n) **Value List Handler**. Implementado coma Beans de sesión, encárgase de manexar a execución de consultas SQL, cachealas e procesar os resultados. Accede directamente a un DAO que se encarga á súa vez de facer a conexión co sistema de información e recuperación dos datos. Unha vez obtidos almacénaos como TO ou VO permitindo ao cliente percorrelos grazas á implementación do patrón Iterador.

o) **Service Activator**. Proporciona un modelo para mensaxería asíncrona como JMS. O Service Activator recibe as mensaxes e localiza e chama aos métodos dos compoñentes de negocio que se van encargar de resolver a petición.

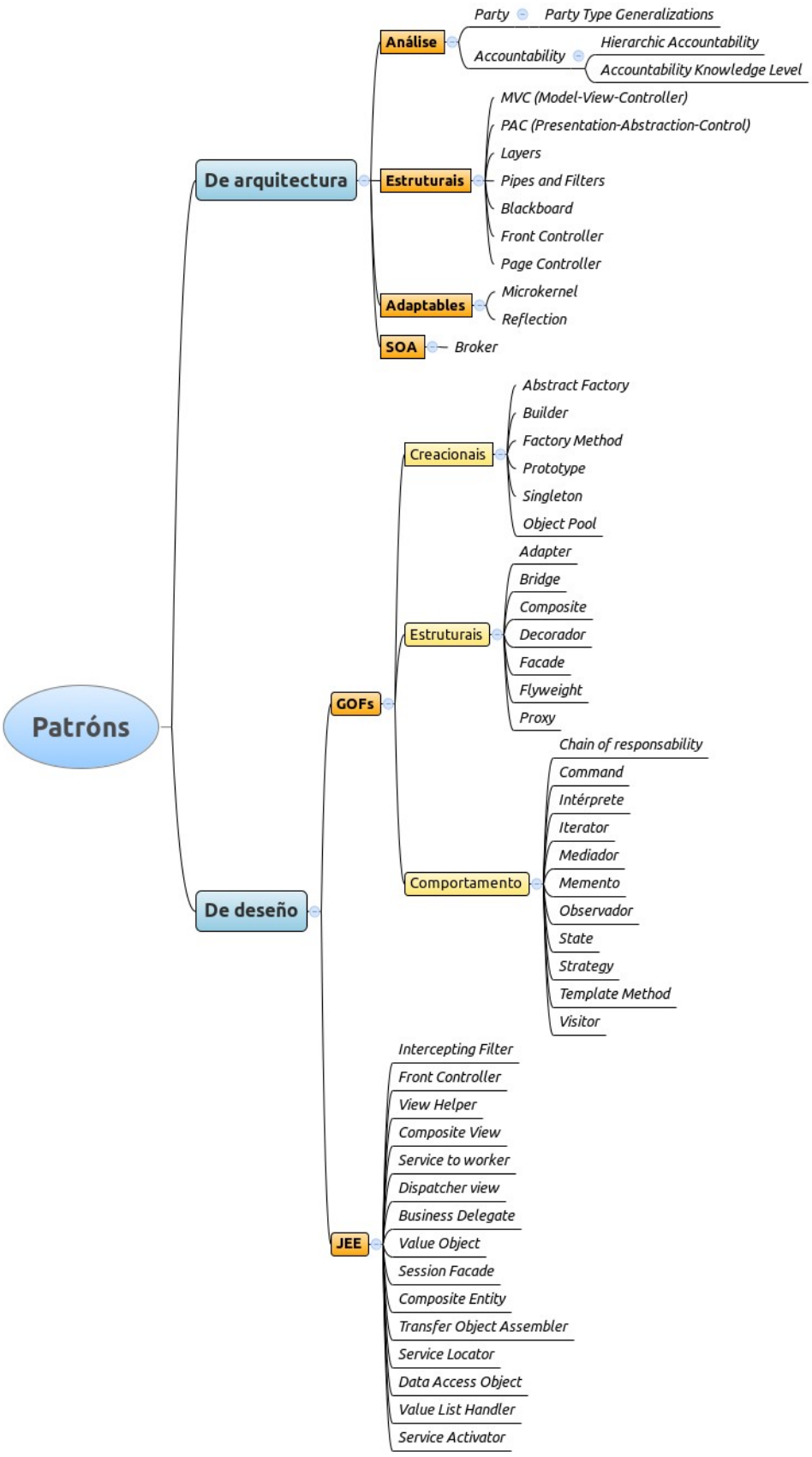


Figura 1: Resumo dos principais patróns en arquitecturas de servidores de aplicacións.

A implementación destes patróns non acostuma a facerse a medida senón que se recorre aos ***frameworks***. Moi relacionados entre si, os *frameworks* representan unha arquitectura de pequeno tamaño que proporciona unha estrutura xenérica integrando diferentes patróns de xeito que poidan ser reutilizados ou integrados de xeito doado nas aplicacións. Nun *framework* os patróns teñen unha implementación concreta sobre a definición abstracta do patrón. En última instancia son un conxunto de clases e interfaces que cooperan para ofrecer un software reutilizable.

31.3 MVC

O patrón Modelo-Vista-Controlador é o máis empregado para estruturar unha aplicación atendendo a unha correcta separación en capas: entrada, procesamento e saída. As súas principais **vantaxes** son unha redución do acoplamento, facilidade de desenvolvemento, claridade no deseño, mellora no mantemento, maior escalabilidade, unha maior cohesión con cada capa fortemente especializada, e unha maior flexibilidade e axilidade nas vistas, permitindo a súa modificación dinámica, sincronización, aniñamento e a existencia de múltiples vistas.

As **capas** do modelo concrétanse en:

1) **Modelo** (en inglés *Model*). Encapsula tanto datos como as funcionalidades ou casos de uso. Ten que funcionar independentemente de calquera representación que tomarán os datos na saída e calquera comportamento que se especifique na entrada do sistema. A todos os efectos será unha caixa negra que recibe peticións de devolve resultados, encargándose de manexar os datos e controlar as súas transformacións. Normalmente implementa os patróns DAO, VO e Fachada.
2) **Vista** (en inglés *View*). Capa na que se integran todos os compoñentes que afecten á interface de usuario. Recibe as peticións do usuario e as envía cara o controlador, obtendo deste as respostas. Permítense múltiples vistas do mesmo modelo, pero toda a lóxica de presentación debe ir nesta capa.
3) **Controlador** (en inglés *Controller*). Recibe peticións da vista, tales como eventos, refrescos, etc... que recolle cun xestor de eventos ou Handler e son traducidos a solicitudes de servizos ou casos de uso, enviando as peticións ao modelo. A miúdo implementan patróns como Comando ou Front-Controller para encapsular as accións.

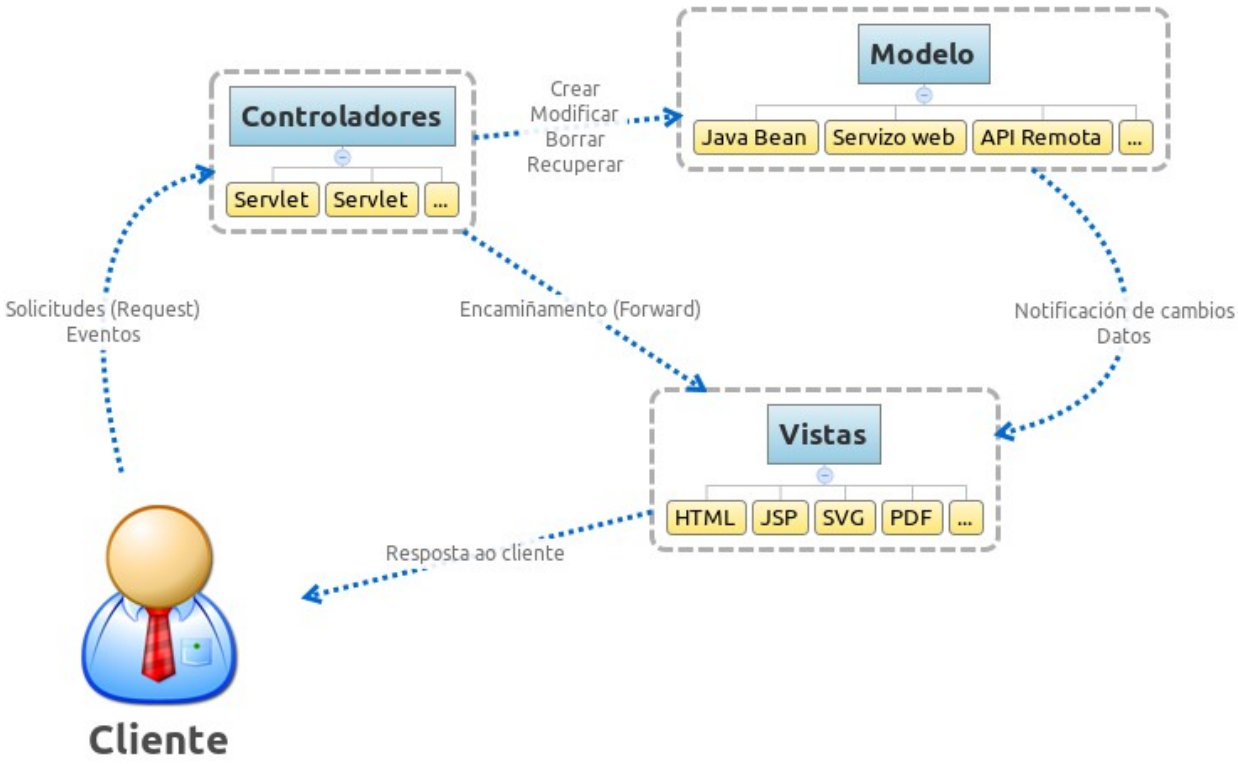


Figura 2: Modelo-Vista-Controlador con tecnoloxías JEE.

Como patrón de arquitectura o MVC pode conter á súa vez os seguintes **patróns** de deseño:

- ✔ **Observador**. Para prover o mecanismo de publicación e subscrición que permita notificar cambios do modelo nas vistas.
- ✔ **Composite View.** Para permitir a creación de vistas compostas nunha xerarquía.
- ✔ **Estratexia**. Para levar conta da relación entre as vistas e os controladores, xa que permite modificar dinamicamente aspectos do control.
- ✔ **Factory Method.** Para especificar ao controlador unha vista coma predeterminada.
- ✔ **Decorador**. Para engadir funcionalidades adicionais ás vistas.
- ✔ **Proxy**. Para distribuír a arquitectura en diferentes emprazamentos e mellorar características de rendemento.

O modelo MVC impleméntase tanto en *frameworks* .NET (Windows Forms, ASP .NET, Spring .NET, Maverick .NET, MonoRail, ...) como en JEE (Struts, Spring, Tapestry, Aurora, JSF, etc..). Así mesmo é un modelo que se atopa estendido a moitas outras tecnoloxías coma PHP, Ruby, Perl, Phyton, etc ...

Os *frameworks* que implementan o MVC acostuman presentar unha serie de **características xerais,** comúns a todos eles e que inclúen:

- ✔ Implementación de diferentes patróns de deseño orientados á reutilización de deseño e código.
- ✔ Controis de validación de campos de formularios.
- ✔ Control de erros e excepcións.
- ✔ Mensaxería e localización de cadeas de textos.
- ✔ Librarías de etiquetas ou compoñentes (TagLibs, Widgets, etc...)
- ✔ Compoñentes da Interface de Usuario como etiquetado de compoñentes de formularios, pestanas, controis AJAX, etc...
- ✔ Presentación de información a través de listados e táboas con paxinación.
- ✔ Integración con *frameworks* co patrón Decorador ou baseados en modelos como Tiles, FreeMarker, Velocity, etc...
- ✔ Acceso datos en diferentes Sistemas de Información: Bases de datos, XML, etc...
- ✔ Abstracción de enderezos URL, Request e sesións.
- ✔ Autenticación e control de usuarios, roles e filtros.

Entre os frameworks que implementan o MVC destaca Apache Jakarta **Struts** (que ten unha evolución en Struts 2.0 ao fusionalo con WebWork), un dos máis empregados en tecnoloxías JEE e que resulta case un estándar de facto debido á súa integración noutros *frameworks* con máis funcionalidades. Emprégase para a implementación de aplicacións web baseadas en Servlets e JSP. Proporciona un conxunto de etiquetas JSP personalizadas (en inglés *Custom Tags*) que permiten encapsular funcionalidades na vista. Co modelo de Struts ten implantación directa o modelo MVC e outros patróns de deseño pre-construídos, permitindo a configuración directa de obxectos reutilizables perante a configuración de XML. Ademais proporciona as características anteriormente especificadas: validación, localización, modelos, etc...

Transporta automaticamente os datos inseridos polo cliente ata o controlador a través de Accións (en inglés *Actions*) mediante formularios ActionForms integrados no *framework* e vice versa para a súa presentación. Distingue entre unha parte común a calquera aplicación que faga uso do *framework* que fai de Controlador (ActionServlet) e outra parte configurable a través de arquivos de configuración en XML (struts-config.xml, web.xml, ...). A súa principal desvantaxe é non abarcar ata o nivel de acceso a datos, facendo que sexa necesario o emprego doutros *frameworks* especializados nesta capa para a elaboración de DAO, VO e outras operación complementarias.

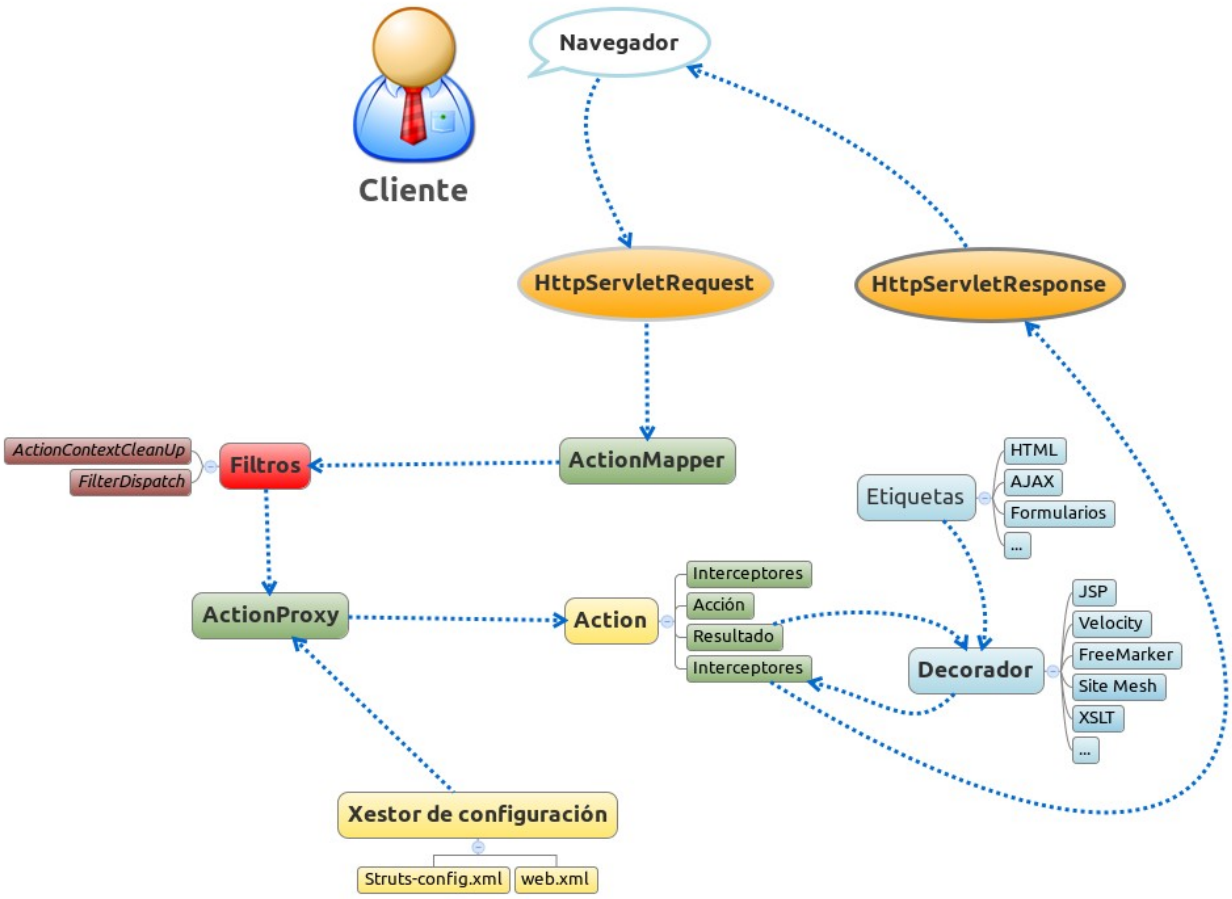


Figura 3: Funcionamento interno Struts/Struts 2.0.

A principal alternativa a Struts sería **Spring** Framework, aínda que tamén permiten integración conxunta e con outros *frameworks* como JSF, Tapestry ou WebWork. Aínda que a súa orientación principal sexa a plataforma JEE, está dispoñible en .NET a través do *framework* Spring .NET. Ten soporte para JTA, JDO, JDBC e ODBC, e permite integración con terceiros como Acegi, Hibernate, iBatis e OJB. Como novidade permite programación orientada a aspectos ou AOP (en inglés *Aspect-Oriented Programming*) que busca empregar os servizos secundarios como seguridade, rexistro de log, manexo de transaccións, etc... das funcionalidades do modelo. Con AOP poden empregarse os servizos da aplicación de forma declarativa, ou perante arquivos XML de configuración ou mediante estándares JSR. Así mesmo realiza Inversión de Control ou IoC, que promove o baixo acoplamento a partir da inxección de dependencias entre obxectos. As principais desvantaxes de Spring son que implica unha configuración complexa, xa que cada servizo leva o seu XML propio, aínda que existe a alternativa do JSR. O seu contedor non resulta lixeiro o que impide que teña aplicación práctica nalgúns contornos como poden ser os dispositivos móbiles.

A **arquitectura de Spring** está composta polos seguintes compoñentes:

- ✔ **Core.** O núcleo que aloxa o contedor principal ou BeanFactory.
- ✔ **Módulo AOP**. Prove a implementación de AOP, permitindo desenvolver interceptores de método e puntos de ruptura para desligar o código do modelo das funcionalidades transversais.
- ✔ **Modulo DAO.** Prove a capa de abstracción de acceso a datos e sistemas de información sobre os diferentes conectadores dispoñibles. Ademais prove de manexo de transaccións vía AOP e outros servizos.
- ✔ **Módulo ORM.** Prove integración para as distintas API de correspondencia entre obxectos e entidades de bases de datos con soporte de diferentes tecnoloxías e integración con *frameworks* de terceiros.
- ✔ **Módulo JEE.** Integración con aplicacións e servizos JEE.
- ✔ **Módulo Web.** Aporta compoñentes especiais orientados a desenvolvemento web e integración con *frameworks* alternativos como Struts ou JSF, ademais dunha implementación do paquete Spring MVC.

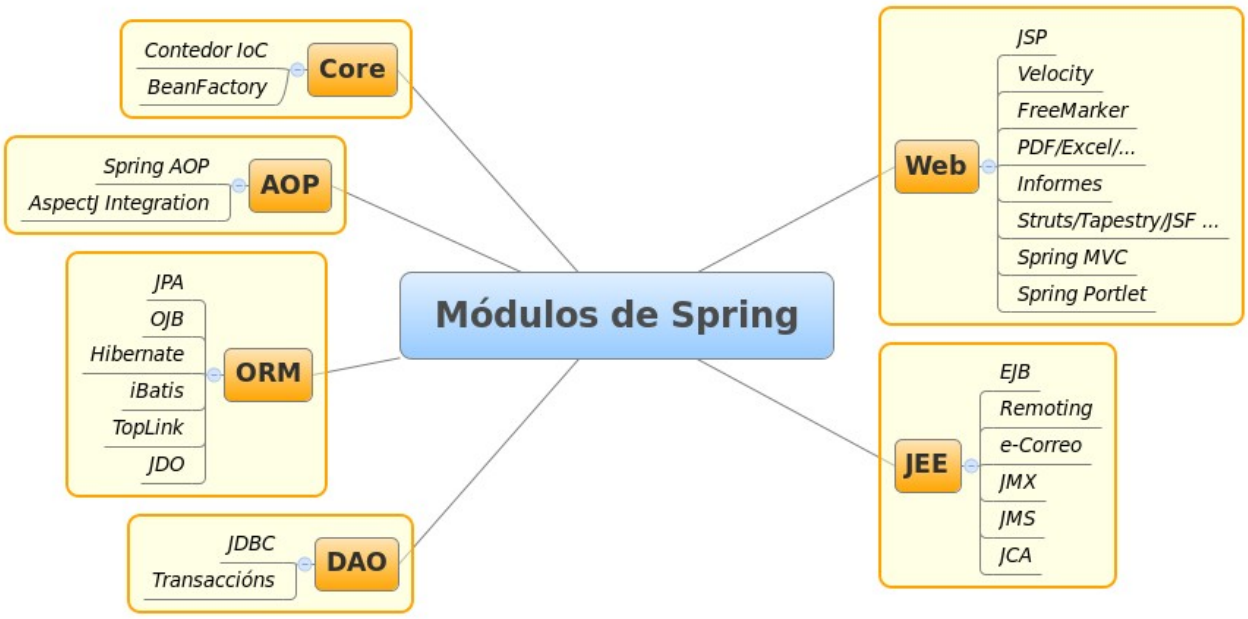


Figura 4: Arquitectura de Spring.

31.4 JSF

A tecnoloxía Java Server Faces proporciona un *framework* de interface de compoñentes de usuarios para o lado do servidor de aplicacións. Na súa base emprega JSP pero permite outras tecnoloxías para interfaces de usuario como XUL. Entres os **compoñentes** de JSF atópanse:

1) Un conxunto de APIs para representar e manexar compoñentes da interface de usuario. Entre as opcións que xestionaría atoparíanse control de estado e eventos, validacións de formularios, conversión de datos, control de navegacións e soporte de localización e accesibilidade.
2) Un conxunto de compoñentes da interface de usuario reutilizables.
3) Dúas librarías de etiquetas personalizadas (en inglés *Custom Tags*) para JSP.
4) Modelo de eventos para o lado do servidor.
5) Soporte para Managed Beans de control de eventos.

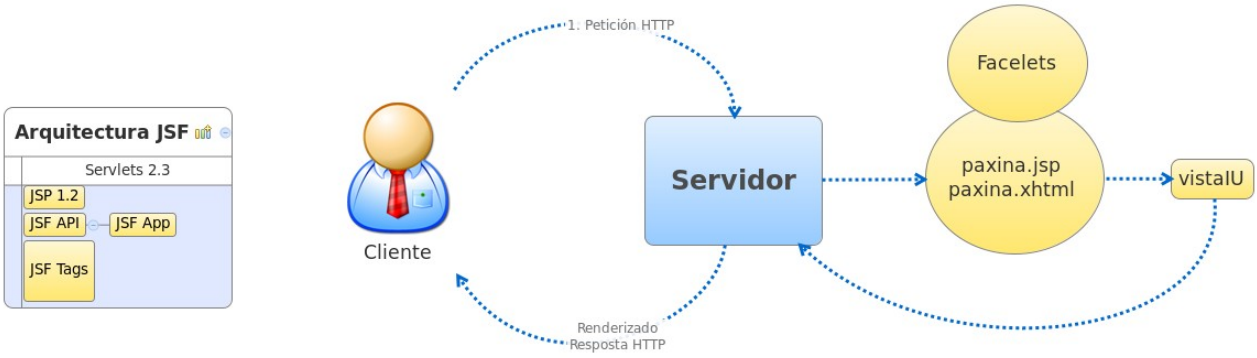


Figura 5: Arquitectura JSF e funcionamento básico.

Un dos compoñentes de JSF é o *framework* JavaServer **Facelets**, destinado á xestión de modelos (en inglés *templates*). As principais características deste *framework* son:

- ✔ Custe de tempo cero para o desenvolvemento de etiquetas de compoñentes da Interface de Usuario.
- ✔ Facilidade de creación de modelos de páxinas e compoñentes reutilizables.
- ✔ Soporte para UEL (en inglés *Unified Expression Language*) e validacións EL.
- ✔ Compatibilidade con calquera RenderKit.
- ✔ Intégrase plenamente con JSTL cousa que en JSF pode ocasionar problemas.
- ✔ Compilación máis rápida que con JSP.

Actualmente existen numerosas **implementacións** de JSF que poden complementar á especificación oficial JEE. Existe a posibilidade de combinar diferentes implementación nunha mesma aplicación, sendo as máis habituais:

a) **MyFaces Tomahawk/Sandbox.** Desenvolvido por Apache proporciona un conxunto de compoñentes reutilizables compatibles coas especificacións JSF 1.1, JSF 1.2 e JSF 2.0.
b) **Trinidad**. Subproxecto de MyFaces, a partir da inclusión dos compoñentes ADF Faces e outras melloras. Prove dos seguintes elementos: Unha implementación de JSF, varias librarías de compoñentes Widgets, a extensión MyFaces Orchestra e módulos de integración para outras tecnoloxías e estándares como MyFaces Portlet Bridge.
c) **Tobago**. Outro proxecto baseado en MyFaces nunha aproximación do deseño de páxinas web ao de aplicacións de escritorio. Proporciona unha serie de compoñentes da Interface de Usuario como abstraccións do HTML. Presenta un conxunto de temas para clientes HTML con vistas independentes de HTML/CSS/Javascript.
d) **ICEfaces**. Contén diversos compoñentes de interfaces de usuario enriquecidas baseadas en AJAX e compatibles con SSL, como editores de texto, reprodutores multimedia, etc... Soporta Facelets e Seam, ademais de ser compatible con Spring, WebWork e Tomahawk.
e) **RichFaces**. Outro *framework* AJAX que inclúe ciclo de vida, validacións, conversións e xestión de recursos nas aplicacións. Soporta Facelets e Seam, ademais de ser compatible con Spring e Tomahawk.

f) **Ajax4JSF**. Outra alternativa máis que proporciona un *framework* AJAX que inclúe ciclo de vida, validacións, conversións e xestión de recursos nas aplicacións. Soporta Facelets e Seam, ademais de ser compatible con Spring e Tomahawk. Inclúe os seguintes compoñentes:

- ✔ *Ajax Filter.* Filtro de peticións para AJAX.
- ✔ *Ajax Action Components.* Envían as peticións dende o cliente.
- ✔ *Ajax Containers.* Interface que describe zonas dentro das JSP.
- ✔ *Javascript Engine.* Motor no lado do cliente que actualiza diferentes zonas das JSP en función da resposta AJAX.

31.5 ANTIPATRÓNS

Contrarios ao **concepto** de patróns, os antipatróns representan malos usos habituais, ou solucións que, sobre todo ao longo do tempo, presentan máis problemas dos que resolven, trátase en definitiva de malas prácticas. Existen dúas variantes principais, os que describen unha mala solución para un problema habitual e que produce consecuencias difíciles de arranxar ao longo do tempo; e aqueles que describen como poñer remedio a un problema e convertelo nunha boa solución. Por norma xeral os antipatróns vense como unha boa idea ao comezo, que falla de mala maneira á hora da súa implementación.

As **motivacións** ou razóns para ter en conta os antipatróns como caso de estudo atenden aos seguintes puntos:

- ✔ Permiten identificar solucións de risco para problemas habituais.
- ✔ Proven experiencia do mundo real para detectar problemas que se repiten ao longo do tempo, ofrecendo posibles solucións ou alternativas para as súas implicacións máis habituais.
- ✔ Proven dun marco común para a identificación e documentación dos problemas e deseño das solucións.

Como acontecía cos patróns, os antipatróns acostuman a agruparse en diferentes **categorías**, sendo as principais:

1) **Antipatróns de desenvolvemento software.** Definen problemas asociados ao desenvolvemento software a nivel de aplicación, ao nivel dos patróns de deseño.
2) **Antipatróns de arquitectura de software.** Céntranse na distribución e relacións das aplicacións, servizos e outros compoñentes software a nivel de organización.
3) **Antipatróns de xestión de proxectos software.** Identifican escenarios críticos sobre a comunicación entre persoas e e resolución de problemas en equipos, vendo como afectan a un proxecto ou proceso software.

Así mesmo os antipatróns teñen aplicación en moitas outras áreas como metodoloxía, xestión da configuración, TDD, deseño web, accesibilidade, usabilidade, etc...

Dentro dos **antipatróns de desenvolvemento** software atopámonos entre os máis comúns:

a) **Blob ou obxecto todopoderoso** (en inglés *God Object*). Emprégase un único obxecto, clase ou módulo para aglutinar un amplo conxunto de funcionalidades que deberían atoparse divididas. Con este patrón cáese nun código amplamente desorganizado e moi acoplado.
b) **Fluxo de lava ou lava seca** (en inglés *Lava Flow*). Representa aqueles tipos de programación por impulsos ou erupcións de código, de xeito desestruturado, desorganizado e con pouca documentación. O sistema medra de xeito desproporcionado e pasado un tempo os bloques de código máis antigos considéranse metaforicamente solidificados no tocante á dificultade de solucionar calquera tipo de problema no que se atopen involucrados.
c) **Descomposición funcional.** Deseño non orientado a obxectos, froito da migración dende linguaxes estruturadas a POO.
d) **Poltergeists**. Ou clases pantasma debido ao descoñecemento dentro da aplicación de cal é o obxectivo dalgunhas clases, sendo en moitos casos súa única función transmitir información entre clases.
e) **Martelo dourado.** Empregar a mesma solución para calquera problema que xorda, sen contemplar outras posibles alternativas.
f) **Código spaghetti.** Fai referencia a código de aplicación cunha estrutura complexa e incomprensible con multitude de tecnoloxías mesturadas. A analoxía faise a partir das relacións entre o código que semellan un grande número de fíos mesturados e enrolados.

g) **Programación copiar e pegar.** Solución na que en lugar de crear solucións xenéricas cópianse e adáptanse solucións xa existentes.

No tocante aos **antipatróns de arquitecturas** software destacan por ser os máis habituais:

a) **Reinventar a roda.** Implementar compoñentes xa dispoñibles ou que poden aproveitarse con lixeiras modificacións. Dáse pola tendencia a facer todo un mesmo ou o descoñecemento da arquitectura e solucións dispoñibles no mercado ou alternativas de código aberto.

b) **Vendor Lock-In.** Construír unha arquitectura dependente dun produto de terceiros, en especial cando se trata de software privativo. Ponse en perigo a escalabilidade do sistema e aumentan os custes de mantemento.

c) **Illamento na organización.** Nunha mesma organización ou conxunto de sistemas créanse diferentes unidades illadas entre si que medran en paralelo solucionando problemas comúns de xeito independente. Neste modelo pode medrar sobre maneira o custe de integración chegada a necesidade do mesmo.

d) **Deseño por comité.** Demasiadas persoas participan dos requirimentos do proxecto dando lugar a un deseño demasiado abstracto e excesivamente complexo por mor de contemplar demasiados puntos de vista particulares. Complícase a toma de requisitos dando a lugar a demasiadas reunións de longa duración, que dificultan e provocan erros ao longo de todo o ciclo de vida de desenvolvemento.

e) **Arquitectura por implicación.** Non existe documentación da arquitectura do sistema, nin dos procesos, nin das tarefas automatizadas máis habituais.

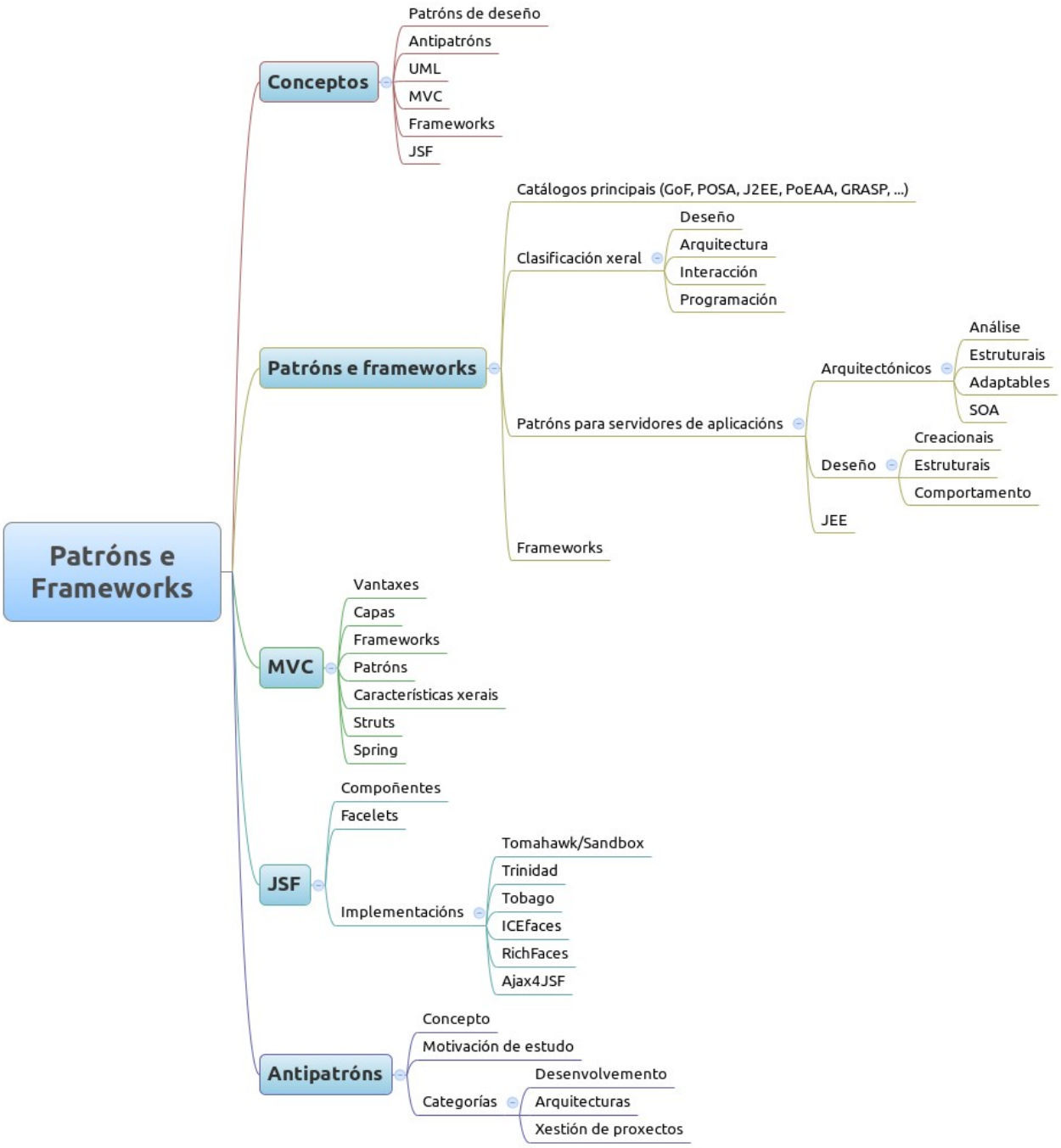
No tocante aos **antipatróns de xestión de proxectos** software destacan por ser os máis habituais:

a) **Parálise de análise.** Os procesos de análise e deseño prolóngase tanto que o proxecto remata morrendo nel sen chegar a implementarse. Son desenvolvementos opostos aos modelos baseados en prototipos e iterativos.
b) **Morte por planificación.** Demasiada planificación e reunións sen chegar a concretar puntos

de partida para o desenvolvemento. De novo son desenvolvementos opostos aos modelos baseados en prototipos e iterativos.

c) **Persoas problemáticas** (en inglés *corncob*). Persoas difíciles de participar en equipos, ou con pouca capacitación ou aptitude, obstrúen, desvían e mesmo sabotan o desenvolvemento.
d) **Xestión irracional.** A falta de decisión e capacitación sumado á nula planificación poden dar lugar á toma de decisións con posterioridade e desenvolvementos de urxencia.
e) **Proxectos sen xestión.** Non se atende ao análise e o deseño, só a implementación. Vanse arranxando incidencias segundo acontecen en modo pila, as últimas primeiro.

31.6 ESQUEMA


Patróns e Frameworks
Conceptos
Patróns de deseño
Antipatróns
UML
MVC
Frameworks
JSF
Patróns e frameworks
Catálogos principais (GoF, POSA, J2EE, PoEAA, GRASP, ...)
Clasificación xeral
Deseño
Arquitectura
Interacción
Programación
Patróns para servidores de aplicacións
Arquitectónicos
Análise
Estruturais
Adaptables
SOA
Deseño
Creacionais
Estruturais
Comportamento
JEE
Frameworks
MVC
Vantaxes
Capas
Frameworks
Patróns
Características xerais
Struts
Spring
JSF
Compoñentes
Facelets
Implementacións
Tomahawk/Sandbox
Trinidad
Tobago
ICEfaces
RichFaces
Ajax4JSF
Antipatróns
Concepto
Motivación de estudo
Categorías
Desenvolvemento
Arquitecturas
Xestión de proxectos

31.7 REFERENCIAS

Deepak Alur e outros.
Core J2EE Patterns. Best Practices and Design Strategies. (2003).

William Crawfor e Jonathan Kaplan.
J2EE Design Patterns. (2003).

Steven Metsker e William Wake.
Design Patterns in Java. (2006).