



Capítulo 2 : Marco Teórico

En este capítulo se darán a conocer algunos conceptos básicos del contexto de este trabajo, con la finalidad de situar al problema dentro de un conjunto de conocimientos. Dentro de estos conocimientos se darán algunos conceptos acerca de patrones de diseño, dentro de ellos el patrón MVC, sus partes y características, después se hablará acerca de los *frameworks*, un breve comparativo entre estos y los patrones de diseño y finalmente se hace un pequeño y breve análisis de algunos otros *frameworks* para web existentes.

2.1 Patrones de diseño

2.1.1 Definición e historia

Para comenzar es importante definir lo que un patrón de diseño es, aunque existen varias definiciones al respecto, un patrón de diseño es una solución de calidad para un problema recurrente de diseño. Pero no son aplicables únicamente en el campo computacional, también existen patrones para varias actividades de la vida cotidiana, aunque con algunas diferencias pero tienen el mismo propósito que en el ámbito computacional, proporcionar una base para poder realizar una actividad, mejorando la calidad del producto que esa actividad de como resultado [Freeman, 2004].

Hay patrones que abarcan las distintas etapas del desarrollo; desde el análisis hasta el diseño y desde la arquitectura hasta la implementación. En el caso de los patrones computacionales un *software* estructurado, modulado posee una mejor calidad y es más sencillo corregir errores, implementar mejoras y actualizaciones, ya que un *software* que posee algún patrón de diseño es más sencillo de modificar que un *software* que no posee en



absoluto un patrón. Pero ¿Cómo se debe escoger el patrón adecuado?, esta es una pregunta un poco difícil de responder ya que la mayoría de las actividades de desarrollo o producción no se ajustan perfectamente a un patrón definido, por eso es importante llevar a cabo un análisis para poder visualizar cual será el patrón que mejor se ajuste a las necesidades de desarrollo. En sí “un patrón de diseño puede verse como una plantilla que puede ser aplicada en muchas situaciones diferentes” [Gamma, 1995], para dar una buena solución.

Los patrones se descubren como una forma indispensable de enfrentarse a la programación a raíz del libro "Design Patterns - Elements of Reusable *Software*" de Erich Gamma, Richard Helm, Ralph Jonson y John Vlissides, a partir de entonces los patrones de diseño que aparecen en ese libro son conocidos como los patrones de la pandilla de los cuatro (GoF, *gang of four*), y comienzan a desarrollarse variaciones y nuevos patrones, en poco tiempo se multiplicaron por 100 y no se limitaban a patrones de diseño sino que cubrían todo lo que se entiende por ingeniería del *software* (desde el análisis hasta la implementación)[Gamma, 1995].

2.1.2 Elementos esenciales

En general un patrón tiene cuatro elementos esenciales:

- **El nombre del patrón:** que se utiliza para describir un problema de diseño, sus soluciones y sus consecuencias, en una palabra o dos. Este nombre ayuda a que sea más sencillo de identificarlo, al hablar o escribir de él e incluso puede dar una idea general o una descripción de dicho patrón.



- **El problema:** describe cuando aplicar el patrón, también puede incluir detalles específicos que se deben cumplir o problemas un poco más detallados, los cuales en conjunto engloban el problema central a solucionar.
- **La solución:** describe los elementos que forman el diseño, sus relaciones, sus responsabilidades y sus colaboraciones. La solución no describe un diseño o implementación en particular ya que un patrón de diseño puede verse como una plantilla que se aplica a un problema específico.
- **Las consecuencias:** son los resultados y desventajas de haber aplicado el patrón. Estas consecuencias implican un impacto en las características del sistema como: flexibilidad, portabilidad y extensión. Además de que ayudan a medir el desempeño del sistema.

2.1.3 Clasificación

El grupo de GoF clasificó los patrones en 3 grandes categorías basadas en su propósito: creacionales, estructurales y de comportamiento [Gamma, 1995].

- **Creacionales:** tratan con las formas de crear instancias de objetos. El objetivo de estos patrones es de abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados.
- **Estructurales:** Los patrones estructurales describen como las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades. Estos objetos adicionales pueden ser incluso objetos simples u objetos compuestos.



- **Comportamiento:** Los patrones de comportamiento ayudan a definir la comunicación e iteración entre los objetos de un sistema. El propósito de este patrón es reducir el acoplamiento entre los objetos.

2.2 Patrón de diseño Model View Controller (MVC)

Una vez establecidas las bases de los patrones de diseño, se puede ya comenzar a hablar más del patrón que se utilizó durante este trabajo: el patrón MVC. Estas son las siglas de *Model View Controller*, en español Modelo Vista Controlador. Esto también se ve reflejado en que cada una de estas palabras representa cada uno de los 3 componentes del patrón MVC. Cada parte juega un rol fundamental para la completa integración del sistema.

2.2.1 Definición e historia

“El propósito de este patrón es simplificar la implementación de aplicaciones de acuerdo a las peticiones de los usuarios y los datos a desplegar” [Harrop, 2005]. La definición un poco más formal sería: MVC es un patrón de diseño de *software* que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos de forma que las modificaciones al componente de la vista, o a cualquier parte del sistema puedan ser hechas con un mínimo impacto en el componente del modelo de datos o en los otros componentes del sistema. Este patrón cumple perfectamente el cometido de modularizar un sistema.

El patrón MVC fue descrito por primera vez en 1979 por Trygve Reenskaug, quién trabajaba en Smalltalk en los laboratorios de investigación de la Xerox. Este patrón se ve frecuentemente utilizado en aplicaciones web, donde la vista es la página HTML y el



código provee de datos dinámicos a la página. Las aplicaciones web complejas continúan siendo más difíciles de diseñar que las aplicaciones tradicionales de escritorio, el patrón MVC se presenta como una solución para ayudar a disminuir dicha complejidad.

2.2.2 Componentes

Los 3 principales componentes del patrón MVC son:

- **Modelo:** Representa los datos que el usuario está esperando ver, en algunos casos el Modelo consiste de *Java Beans*.
- **Vista:** es la responsable de transformar el modelo para que sea visualizada por el usuario, ya sea en un archivo de texto normal o en una página web (HTML o JSP) que el navegador pueda desplegar. En sí el propósito de la vista es convertir los datos para que al usuario le sean significativos y los pueda interpretar fácilmente. La vista no debe trabajar directamente con los parámetros del *request*, debe delegar esta responsabilidad al controlador.
- **Controlador:** es la parte lógica que es responsable del procesamiento y comportamiento de acuerdo a las peticiones (*requests*) del usuario, construyendo un modelo apropiado, y pasándolo a la vista para su correcta visualización. En el caso de una aplicación web Java en la mayoría de los casos el controlador es implementado por un servlet.

2.2.3 Tipos de patrones MVC

Actualmente existen dos tipos de patrón MVC:

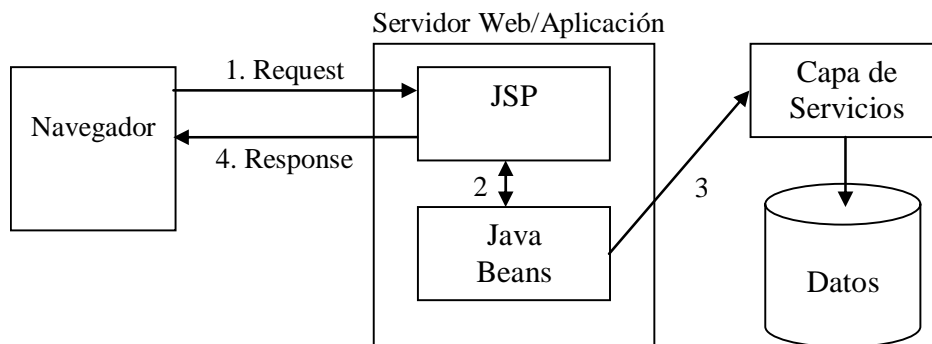


Figura 2.1: MVC de arquitectura Tipo 1 [Walls, 2005]

Como se muestra en la Figura 2.1 en el Tipo 1 de MVC las páginas JSP están en el centro de la aplicación, y contienen tanto la lógica de control como la de presentación. Este tipo de arquitectura funciona de la siguiente manera: el cliente hace una petición a una página JSP; se construye la lógica de la página, generalmente en objetos Java o como se les conoce en Inglés *Plain Old Java Objects* (POJOs) y se transforma el modelo para ser desplegado una vez más.

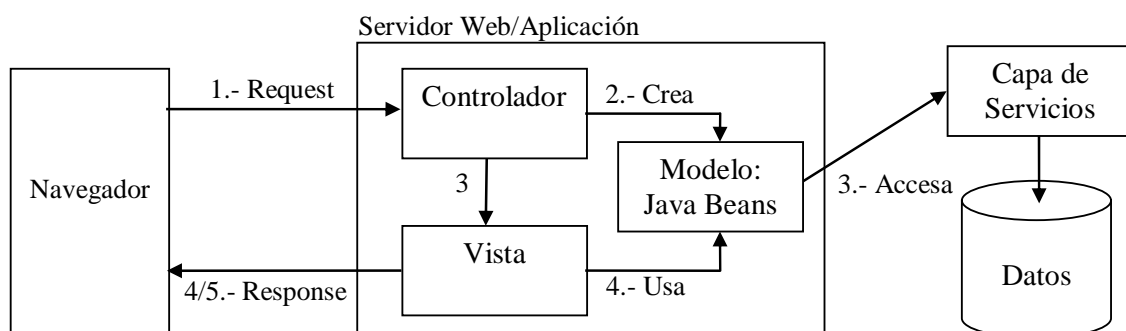


Figura 2.2: MVC de arquitectura Tipo 2 [Walls,2005]



En el modelo de Tipo 2 de MVC, que se aprecia en la Figura 2.2, se puede observar que ya existe una clara separación entre el controlador y la vista, ya que ahora es directamente el controlador quien recibe la petición, prepara el modelo y lo transforma para que sea desplegado en la vista. Este tipo de arquitectura MVC es el que se utiliza para aplicaciones más complejas, ya que para una aplicación sencilla puede utilizarse el Tipo 1. La tecnología JSP no es la única que se puede emplear para las vistas, existen otro tipo de tecnologías que pueden servir como vistas.

2.3 Frameworks

Un *framework* es un término utilizado en la computación en general, para referirse a un conjunto de bibliotecas, utilizadas para implementar la estructura estándar de una aplicación. Todo esto se realiza con el propósito de promover la reutilización de código, con el fin de ahorrarle trabajo al desarrollador al no tener que rescribir ese código para cada nueva aplicación que desee crear. Existen diferentes *frameworks* para diferentes propósitos, algunos orientados al desarrollo de aplicaciones web, otros para desarrollar aplicaciones multiplataforma, para un sistema operativo o lenguaje de programación en específico, entre otros.

Según Gamma, el *framework* determina la arquitectura de una aplicación [Gamma, 1995]. Este es un buen enfoque, ya que el *framework* se encarga de definir la estructura general, sus particiones en clases y objetos, las responsabilidades clave, así como la colaboración entre dichas clases y objetos. Todos estos parámetros son definidos por el *framework*, evitando que el usuario tenga que definirlos y se pueda enfocar en cosas específicas de su aplicación.



“El *framework* captura las decisiones de diseño que son comunes a su dominio de aplicación” [Gamma, 1995]. Un *framework* no sólo promueve la reutilización de código sino también la reutilización de diseño.

Un *framework* ayuda a que se desarrolle una aplicación de una manera más rápida, ya que se no pierde tiempo en algunos detalles de diseño que muchas veces quitan más tiempo del que tomo construir en sí la lógica de la aplicación. Además las aplicaciones que se construyen tienen estructuras similares, son más fáciles de mantener y consistentes para los usuarios. Pero esto tiene como consecuencia una mínima pérdida de libertad en las cuestiones de diseño.

En algunas ocasiones un desarrollador tiene algunas dificultades para diseñar una aplicación, esto todavía es más difícil para el desarrollador del *framework*, ya que desarrollar un *framework* requiere de muchos cuidados, porque cuando se lanza uno nuevo, se espera que pueda servir para muchos tipos de aplicaciones pero con arquitecturas y requerimientos similares. Es decir trata de englobar toda una gamma de aplicaciones dentro de un solo estándar, lo cual puede ser el éxito o el fracaso del mismo, por eso se intenta crear *frameworks* lo más extensibles y flexibles posibles, para que con algunos cambios mínimos se pueda actualizar o corregir. Las aplicaciones que se desarrollan a partir de un *framework*, está ligada al mismo, por eso las aplicaciones deben de evolucionar y crecer al mismo tiempo que crece el *framework*, ya que un cambio en alguna interfaz del mismo significará un cambio de la aplicación, dependiendo de que tan drástico sea el cambio.



2.3.1 Relación entre patrones de diseño y *frameworks*

Los *frameworks* utilizan un variado número de patrones de diseño, ya que así logran soportar aplicaciones de más alto nivel y que reutilizan una mayor cantidad de código, que uno que no utiliza dichos patrones. “Los patrones ayudan a hacer la arquitectura de los *frameworks* más adecuada para muchas y diferentes aplicaciones sin necesidad de rediseño” [Gamma, 1995]. Por esta razón es importante que se documenten que patrones utiliza el *framework* para que los que se encuentren familiarizados con dichos patrones puedan tener una mejor visión y poder adentrarse en el *framework* más fácilmente.

Aunque muchas personas cometen el error de confundir a los *frameworks* con los patrones de diseño, según los cuatro autores del libro “Design Patterns - Elements of Reusable” existen 3 diferencias fundamentales entre ellos [Gamma, 1995]:

- **Los patrones de diseño son más abstractos que los *frameworks*:** el código del *framework* se escribe una vez, en cambio cada vez que se requiere un patrón de diseño se debe de codificar con respecto a la aplicación. Una fortaleza de los *frameworks* es que pueden ser escritos en un lenguaje de programación, pueden ser reutilizados e incluso ejecutados, esto también podría considerarse una debilidad dependiendo del punto de vista (siempre y cuando no sean multiplataforma o se encuentre en diferentes versiones por lenguaje de programación), ya que están enfocados a un sólo lenguaje de programación, en cambio un patrón de diseño puede ser aplicado a cualquier lenguaje de programación, pero se tiene que reescribir desde cero.



- **Los patrones de diseño son elementos arquitectónicos más pequeños que los *frameworks*:** un solo *framework* contiene varios patrones de diseño, pero jamás es al contrario.
- **Los patrones de diseño son menos especializados que los *frameworks*:** existen *frameworks* con un dominio específico, además los patrones son un poco más generales debido a que pueden ser utilizados en un mayor número de aplicaciones de varios tipos.

2.4 *Frameworks* para aplicaciones web

Actualmente existen algunos *frameworks* para desarrollar aplicaciones web, que es de las ramas más importantes en las que se usan los *frameworks*. La mayoría de ellos utilizan el patrón de diseño MVC del cual se habló previamente. Todos los *frameworks* tienen características especiales que los hacen únicos para sobresalir y poder seguir en el mercado, además de poseer las siguientes características comunes [Johnson, 2003]:

- Utilizan un solo servlet que tiene la función de controlador, para toda la aplicación o gran parte de ella. Se configura el *deployment descriptor* “web.xml” para que todas las URL’s tengan que pasar forzosamente por dicho servlet.
- Una configuración, generalmente escrita en un archivo XML, en donde se le indicará al servlet controlador, a través de propiedades, a quien delegar la responsabilidad de atender la petición entrante. Algunas veces esas propiedades están indicadas de acuerdo a los URL’s y de acuerdo al URL entrante es como se delega la responsabilidad.



- Las vistas pueden tener nombres claves, sin necesidad que exista una relación con el nombre del archivo de la vista. El *framework* se encarga de realizar dicha conversión para poder obtener el nombre de la vista que se tiene que cargar para que sea desplegada. La implementación de una vista con un nombre en particular puede cambiar sin afectar código del controlador.

Cuando se va a desarrollar una aplicación web, el desarrollador se debe fijar en si desea realizar una aplicación extremadamente sencilla o si quiere desarrollar una verdadera aplicación web, que tendrá actualizaciones, correcciones y mejoras a futuro. Para esto es recomendable que se elija un *framework* de aplicación web que utilice el patrón web MVC. Con el fin de que se pueda hacer la separación entre los 3 elementos principales y pueda aprovechar todas las ventajas que brinda este patrón de diseño. Para poder aprovechar también las ventajas adicionales que brinda el *framework* para la integración con otras herramientas u otros servicios.

A continuación se enlistan algunos de los *frameworks* para aplicaciones web más populares: Struts, WebWork, Maverick y Spring, el *framework* sobre el cual se basa esta tesis, será analizado con mayor detalle en el Capítulo 4 de este documento. Este listado y análisis sencillo de cada uno de estos *frameworks* se da con la razón de poder saber cuales son los *frameworks* contra los que compite Spring, dentro del mismo ámbito, así como sus características principales, ventajas y desventajas.

2.4.1 Struts

Struts es uno de los *frameworks* MVC más utilizados, ya que fue uno de los pioneros en el campo, fue creado por Craig McClanahan, creador del famoso motor servlet



Tomcat, ambos son distribuidos por Apache. Este *framework* fue lanzado a mediados del año 2000, y a partir de esta fecha comenzó a tener popularidad, y en la actualidad existen algunos componentes extra que se adaptan a Struts, lo que le da un mayor campo de acción. Struts es *open source*, por lo que no requiere licencia para su uso [Johnson, 2003].

Además cumple una de las funcionalidades básicas que se mencionaron acerca de los *frameworks* para aplicaciones web, ya que implementa un solo controlador (ActionServlet) que evalúa las peticiones del usuario mediante un archivo configurable (struts-config.xml) para poder dirigirlos a un *action* que procesa el *request*. El lenguaje de programación que utiliza es Java, así como Java Beans como modelo.

Algunas de las ventajas que Struts brinda son: que existe un variado número de trabajos y proyectos ya hechos lo que brinda un mayor número de ejemplos para poder tomar un punto de partida y de referencia; otra ventaja es que brinda librerías de *tags* para HTML bastante útiles. Entre las desventajas de *framework* se puede observar que muchas veces puede ser difícil trabajar con los ActionForms, además de que el proyecto posiblemente desaparezca en los años venideros, otra de las desventajas de Struts es que muy ligado con la tecnología JSP, por lo que muchas veces se dificulta integrarlo con alguna otra tecnología para las vistas.

2.4.2 Maverick

Este es otro *framework* MVC *open source* que existe en el mercado, pero a diferencia de los demás este no cuenta con sus propias librerías de *tags*. Sin embargo cumple con las funcionalidades típicas mencionadas, como el de tener un solo servlet controlador central como punto de entrada, el cual lleva el nombre de *Dispatcher*, que está



definido en el *Deployment Descriptor* de la aplicación web (web.xml). Maverick cuenta con un archivo XML en el que se guarda toda la configuración del mismo (maverick.xml). Una característica de Maverick es que únicamente acepta un solo controlador central y un archivo de configuración por aplicación web, lo que muchas veces al desarrollar aplicaciones más grandes y complejas se puede volver confuso y difícil de configurar.[Johnson, 2003]

Maverick es un *framework* mucho más configurable que Struts, lo que brinda cierta flexibilidad, también incluye varias clases para poder extender y cambiar el flujo del trabajo o *workflow* en Inglés. Maverick es usualmente usado para crear nuevos controladores que sean capaces de procesar nuevas peticiones. Los controladores son Java Beans, y el *framework* pone de manera transparente las propiedades de los *beans*.

Una característica interesante de Maverick es el básico y fácil uso de la funcionalidad del *dispatcher*. También es multiplataforma ya que ha sido adaptado para los lenguajes .NET y PHP.

2.4.3 WebWork

Este es un *framework* más reciente ya que fue lanzado a mediados del año 2002, una de las personas que ha trabajado en este proyecto fue Rickard Oberg, quien ha participado en proyectos como JBoss, entre otros. “WebWork, a pesar de su nombre, no es puramente un *framework* para aplicaciones web. Adopta un acercamiento práctico que minimiza la dependencia del código de la aplicación de los conceptos web.”[Johnson, 2003].



El *framework* WebWork está basado en el patrón de diseño de Comandos o *Command Pattern*. Cada acción es un comando, que es creado para manejar un *request*, sus propiedades son puestas de manera transparente, ya que cada acción es un Java Bean.

WebWork cuenta, al igual que Struts y Spring con su propia librería de *tags* para JSP, las cuales ayudan a realizar distintas tareas de una manera más ágil, pero no es la única tecnología para vista que soporta, también incluye soporte para Velocity.

Una de las ventajas de WebWork es que cuenta con una arquitectura simple y las clases son fáciles de extender. Pero como todo, tiene sus desventajas, la creación de una acción (*action*) por cada *request* puede llegar a ser algo confuso cuando no se tiene muchos datos en el *request*; impone el patrón de diseño Command en cada interacción del usuario, ya sea bueno o malo utilizarlo en dicha interacción; es difícil saber de que tipo son las excepciones que se lanzan. Como este *framework* es relativamente reciente no existe mucha documentación y ejemplos al respecto lo que puede resultar a veces frustrante cuando se requiere buscar algún ejemplo o tutorial que pueda servir.

2.4.4 Spring

Es un *framework* de aplicación, que a diferencia de otros *single-tier* como Struts, Spring propone estructurar toda una aplicación de una manera consistente y productiva, conjuntando lo mejor de cada uno de los *frameworks single-tier* para crear una arquitectura coherente. En sí Spring ha surgido como una solución para poder disfrutar de los beneficios clave de J2EE, mientras que minimiza la complejidad encontrada en el código de la aplicación.



Spring esta basado en la filosofía de que un *framework* debe proveer una guía hacia una buena práctica, es decir debe hacer la cosa correcta sencilla de hacer. Mezclando la correcta combinación de flexibilidad y restricción, la cual es la clave en el buen diseño de un *framework*.

Spring también cuenta con algunas de las características generales de los *frameworks* web, ya que cuenta con un módulo para poder desarrollar aplicaciones web de manera sencilla, utilizando el patrón de diseño MVC antes mencionado. Contando con un controlador central (DispatcherServlet), además de la capacidad de tener varios controladores secundarios para el procesamiento de cada una de las peticiones.

También se basa en el diseño de las clases extendiendo o implementando interfaces, lo cual es un mejor diseño orientado a objetos, ya que promueve la reutilización de código. Además Spring cuenta con algunos *tags* para las diferentes tecnologías para vista que existen. Las cuales tienen funciones diferentes como vincular formularios, validación, despliegue de información, entre otras.

Uno de los aspectos clave y ventajas de Spring, es que cuenta con una arquitectura modularizada, y se pueden utilizar cada uno de estos módulos de manera independiente. Cada uno esta enfocado en una tarea específica, y algunos de ellos son para la integración con alguna herramienta o incluso cuenta con la posibilidad de integrarse con los otros *frameworks* mencionados anteriormente. Esto con el propósito de proponer la filosofía de “no intentar reinventar la rueda”, la cual quiere decir que si ya existe en el mercado una herramienta que realice una tarea de manera eficiente en un ámbito o área en particular se debe conjuntar con Spring para llevar a cabo un mejor desempeño y poder sacar así un mejor resultado y una aplicación de mayor calidad.



En conclusión cada *framework* tiene sus ventajas y desventajas, escoger uno es una decisión del desarrollador, dependiendo de sus necesidades y requerimientos, del tamaño del proyecto a desarrollar, así como de su experiencia con tecnologías existentes que se integren fácilmente con el *framework* que haya elegido. Esto con el fin de aprovechar algunas tecnologías que estén enfocadas en una área específica, para que brinde una mejor y más sencilla forma de llevar a cabo la tarea. Esta fue una de las razones por la cual se decidió incursar dentro de las características de Spring, y después de un análisis se llegó a la conclusión de que es un buen *framework* que cumple con la mayoría de las características, de las cuales otros *frameworks* escasean.

Una de las cuestiones que ha hecho a Spring evolucionar de manera muy rápida en tan poco tiempo, es que se han incrementado de una manera increíble el número de proyectos que utilizan esta herramienta a través del último año. Esto conlleva a que la cantidad de personas interesadas en este *framework* aumente, así como el soporte, el número de foros de opinión y la publicación de un mayor número de libros y tutoriales.