Trabajo de Grado:

Extensión del MVC para resolver problemas de Hipermedia Física

Alumnas: Challiol, Cecilia Nº 4354/5
De Cristófolo, Valeria Soledad Nº 4365/9

Director: Dr. Gustavo Rossi
Codirectora: Dra. Silvia Gordillo

Índice

Capítulo 1 Introducción

1.1 Introducción a la Tesis

En los últimos años ha crecido la posibilidad de acceso a los dispositivos móviles. Una gran parte de la población cuenta con al menos un dispositivo móvil. La demanda de los mismos, ha generado un aumento de las funcionalidades disponibles en estos.

Como no podía ser de otra forma, en plena era digital, y con el auge de Internet, las unidades móviles comienzan a incorporar diversos tipos de servicios, entre los que se puede mencionar la navegación web. El beneficio evidente que esto trae aparejado es un aumento inherente de la disponibilidad de la información que puede recibir el usuario.

Por ejemplo si una persona se encuentra realizando un tour y pasa frente a un edificio histórico, esto podría generar una inquietud de búsqueda de información sobre el mismo en el dispositivo móvil. En cambio si se encuentra en su casa tal vez nunca le surja la iniciativa de búsqueda sobre ese tema.

Aquí queda al descubierto que el ambiente que rodea a la persona que está usando el dispositivo móvil influye en las inquietudes que motivan a la navegación. Esto nos lleva a tener en cuenta la gran importancia e influencia que tiene el mundo que nos rodea en la utilización de los dispositivos móviles.

Una de las funcionalidades que está en desarrollo para las unidades móviles, es la de permitir al usuario del dispositivo, interactuar no sólo con el dispositivo a través de la navegación, sino también incorporar el espacio físico que lo rodea. La incorporación del espacio físico en las aplicaciones de artefactos móviles, implica tener en cuenta la ubicación de la persona, para poder brindarle navegación e información acorde a lo que la persona está viendo.

La necesidad de agregar información física a la Hipermedia Tradicional, que sólo abarca las relaciones entre los elementos digitales, resultó en la Hipermedia Física.

Es evidente que es mucho más atractivo aumentar el mundo digital incluyendo los objetos físicos que seguir manejando la información digital como un mundo virtual totalmente separado de la realidad. Hace tiempo que se están desarrollando varias investigaciones sobre este tema, y es aquí donde se abren varias ramas interesantísimas de análisis y estudio.

Como resumen de los avances en este área, cabe destacar que todo comienza con la combinación del ambiente real con el virtual, denominado “Realidad aumentada”. En lugar de sumergir a las personas en un mundo virtual artificialmente creado, la meta es aumentar los objetos del mundo físico reforzándolos con una riqueza de información digital y capacidades de comunicación.

El aspecto más innovador de la realidad aumentada no es la tecnología: es el objetivo. Se puede aumentar la realidad a partir de tres estrategias: aumentar al usuario, aumentar los objetos físicos y aumentar el ambiente que rodea al usuario y a los objetos. A modo de ejemplo y para clarificar las ideas podemos pensar en el manejo de documentos. En lugar de intentar reemplazar los documentos con imitaciones electrónicas de ellos, se podrían crear "documentos interactivos" que se unen directamente con aplicaciones en línea.

El problema de "registrar" los objetos del mundo real y machearlos con información electrónica correspondiente, es un área activa de investigación en la realidad aumentada. De este concepto surgen investigaciones como son HyperReal[43] y Augmented Reality[33]

Otro avance en este área son los sistemas de información basados en ubicación, la idea básica es conectar partes de información digital a una coordenada de latitud-longitud específica vía algún dispositivo móvil. Luego, los usuarios desde sus unidades móviles, acceden a esta información. De

esta manera, conseguirán la impresión de que la información digital realmente se halla en cierto modo en ese lugar. Por ejemplo: graffiti, carteles, etc, introducidos en GeoNotes[15].

Otro concepto que surge en este campo son los sistemas de Hipermedia espacial, los cuales pueden pensarse como utilizar un espacio 2D para ordenar información. Del concepto de Hipermedia espacial en 2D se derivan aplicaciones en 3D las cuales se denominan Geo-Spatial Hypermedia, esto se puede ver en un prototipo denominado TOPOS [25]. Este prototipo permite la manipulación y mantenimiento de relaciones espaciales entre materiales en un ambiente tridimensional. Se integra con aplicaciones para soportar colaboración en tiempo real. Esta es una manera de aumentar la realidad multidimencionalmente.

La Hipermedia se basa e implementa bajo el paradigma orientado a objetos, permitiendo construir aplicaciones de realidad mixtas. Si se desea desarrollar Hipermedia como apoyo para crear y mantener las relaciones entre los materiales físicos y digitales, se necesitan analizar los diferentes tipos de relaciones que tendrían sentido en el mundo mixto. Esto se introduce en "Physical Hypermedia"[24] donde se define la Hipermedia Física como un formalismo para construir las aplicaciones de realidad aumentada. Este concepto fue refinado en [26], donde se presenta HyCon que trata de extender Hipermedia con el mundo físico.

El término Hipermedia Física ha sido utilizado para describir cualquier sistema de Hipermedia que trata con las propiedades del mundo físico.

Como es de esperar, la integración de las computadoras a la vida cotidiana, implica forzosamente el análisis y estudio de nuevos modelos y paradigmas. La introducción de nuevos dispositivos y tipos de medios de comunicación llevan a la necesidad de crear nuevas maneras de estructurar la información. Es evidente que dicho análisis y estructuración se vuelve fundamental si a esto le sumamos la integración del espacio físico en el plano de las aplicaciones, teniendo en cuenta situaciones y preferencias de los usuarios.

Nuestra tesis se basará en buscar una solución a las aplicaciones de Hipermedia Física, utilizando conceptos ya estudiados para la Hipermedia Tradicional, tratando de reutilizar desarrollos e implementaciones ya probadas. Además, trataremos de aplicar patrones de diseño, los cuales nos asegurarán una implementación más eficiente, debido a la posibilidad de encontrar estructuras que puedan ser aisladas, estudiadas y reutilizadas. Nuestra intención es aprovechar todos los beneficios ya establecidos para la Hipermedia y para el desarrollo de aplicaciones, para poder brindar recursos que faciliten la realización de aplicaciones en el contexto de la Hipermedia Física.

Hoy en día el ejemplo más común de Hipermedia son las aplicaciones web. A la hora de desarrollar una aplicación web, se debe decidir entre sí se utiliza un *framework web* o si se van a implementar manualmente todas sus funcionalidades. La mayor parte de estos frameworks implementan el modelo MVC y basan la creación de aplicaciones en dicho modelo. De más está aclarar, que los *frameworks* no basados en MVC es preferible descartarlos, ya que independientemente de su eficacia, aumentan las dependencias. El modelo MVC permite una separación casi perfecta entre lo que se conoce como modelo (ó la lógica de negocio), el controlador y la vista.

Debido a que los cambios de requerimientos y especificaciones son una costumbre clásica en cualquier desarrollo, será mucho más fácil migrar una aplicación realizada en un framework MVC a otro framework de similares características, que intentar migrar una aplicación creada con un *framework spaguetti* que no siga ningún patrón de diseño claro.

Basaremos nuestra tesis en los diseños de frameworks para Hipermedia, que implementan MVC, con la intención de conseguir una extensión de los mismos para soportar Hipermedia Física, que permita integrar la ubicación del usuario.

1.2 Contribuciones

Las contribuciones que se podrían desprender de esta tesis son:

- Principalmente la extensión del concepto MVC para aplicaciones de Hipermedia Física.
- En segundo término descripciones de patrones de diseño que se puedan utilizar en aplicaciones de Hipermedia Física.
- Por último un análisis de los "roles" que se desprenden de cada uno de los componentes de la extensión del MVC en una aplicación de Hipermedia Física.

Se esperan estas tres contribuciones antes mencionadas. En primera instancia se trata de proveer a los desarrolladores de aplicaciones de Hipermedia Física, una extensión del MVC para poder soportar ubicación ("MVC con ubicación"). De la misma manera que el MVC simplifica la construcción de aplicaciones de Hipermedia Tradicional, esta extensión tiene por objetivo principal disminuir el esfuerzo necesario para construir aplicaciones de Hipermedia Física.

La segunda contribución es brindar soluciones correctas para este tipo de aplicaciones. Para esto se especificarán varios patrones de diseño aplicables aprovechando las implementaciones ya analizadas para los mismos. Puede ocurrir que haya patrones que tengan que ser ajustados para contemplar las características propias de la Hipermedia Física. Esto está contemplado en nuestra tesis, y se describirán y justificarán las modificaciones necesarias.

La última contribución es lograr que los desarrolladores, que utilicen para diseñar sus aplicaciones el "MVC con ubicación" conozcan el comportamiento de cada uno de los componentes. Dicho conocimiento es esencial para un mejor desarrollo de sus aplicaciones. Para lograr esto se especificarán los "roles" que pueden tener cada uno de los componentes del "MVC con ubicación". Cabe aclarar que los mismos van a depender de donde o hacia donde va él estimulo de feedback, haciendo que varíe el comportamiento de cada uno.

1.3 Estructura

El resto de la tesis se organiza de la siguiente manera: en el Capítulo 2 realizaremos una primera aproximación al tema de la Hipermedia y de la Hipermedia Física, para establecer el ambiente de nuestro trabajo. El capítulo 3 además de introducir los conceptos esenciales del MVC y su funcionalidad, aportará una visión conceptual de los nuevos componentes que aparecen al extender el MVC para soportar Hipermedia Física. Será en el capítulo 4 donde se expongan todos los detalles de implementación del nuevo framework desarrollado. Para ello se explicarán los conceptos introductorios del MVC para web: Struts, que es la base de nuestro framework, y luego abarcaremos todos los nuevos componentes que aparecieron. Terminando el capítulo se resumen los patrones de diseño que recomendamos como buena práctica, para la instanciación de nuestro framework y se discuten los roles de los componentes que se desprenden del mismo. Un ejemplo concreto será presentado en el capítulo 5. Las comparaciones de nuestro trabajo con el realizado en otros campos de la Hipermedia se incluirá en el capítulo 6, y se terminará nuestra tesis con las conclusiones alcanzadas en el último capítulo.

Capítulo 2 Introducción al tema

2.1 ¿ Qué es Hipermedia?

La Hipermedia surge a partir de unir dos conceptos a saber: Hipertexto y Multimedia.

2.1.1 Hipertexto

El Hipertexto provee un método no secuencial de representación y acceso a información. En un documento de Hipertexto la información se encuentra almacenada en una red de nodos conectados por enlaces (*links*). La selección de un enlace de Hipertexto nos permite saltar a otra parte del documento o incluso a otro documento.

El Hipertexto se puede ver como una red cuyos nodos representan la información fragmentada y los arcos son las relaciones que existen entre los fragmentos.

Los elementos que forman el Hipertexto son: Nodos, Links y Anchors. El link constituye la relación que existe entre los nodos. Y el Anchor es una marca que se encuentra en el nodo para indicar la ubicación de un link.

El Hipertexto sirve para representar aplicaciones que tengan gran volumen de información y que generalmente los usuarios sólo necesitan una parte de la información disponible. Un requisito fundamental es que la información debe estar fragmentada y que los fragmentos tengan relaciones entre sí.

Un ejemplo de documento multidimensional podría ser una enciclopedia que se lee sólo por pequeñas partes y las partes que se van a leer se seleccionan mediante un índice por el cual se ha entrado a la enciclopedia. En lo que se refiere a la enciclopedia, el orden de lectura suele estar controlado por una asociación de ideas del lector.

A continuación citamos algunas definiciones de Hipertexto:

- "... escritura de texto no secuencial que se ramifica y permite opciones al lector,…es una serie de fragmentos de texto conectada por enlaces que le ofrecen diferentes caminos al lector ...". [39]

- "...como el uso del computador que trasciende la linealidad, límites y calidad fija de la tradicional forma de escritura de texto". [32]

- "...son ventanas, en una pantalla, las cuales son asociadas a objetos en una base de datos, y enlaces provistos entre estos objetos, tanto gráficamente (íconos etiquetados) como en la base de datos (apuntadores)". [9]

- "... La existencia de una *liga* o *lugar* en cualquier parte de un texto almacenado en la computadora que *vincule* dicho documento con otro lugar en el mismo o en diferente texto, el *acceso* será rápido y facilitado por *botones* o cualquier otra *herramienta* para una *navegación no-lineal*". [28]

En publicaciones menos formales se afirma:

- " Hipertexto, en el nivel más básico, es un manejador de base de datos que permite conectar pantallas de información usando enlaces asociativos. En un nivel mayor, Hipertexto es un ambiente de software para realizar trabajo colaborativo, comunicación y adquisición de conocimiento. Los productos de este software emulan la habilidad del cerebro para almacenar y recuperar información haciendo uso de enlaces para un acceso rápido e intuitivo". [18]

- "Una base de datos que tiene referencias cruzadas y permite al usuario (lector) saltar hacia otra parte de la base de datos, si éste lo desea". [4]

2.1.2 Multimedia

Multimedia es un concepto abierto y polivalente. Se utiliza este término tanto para definir una tecnología como para referirse a un medio de comunicación. Implícitamente al hablar de multimedia nos referimos a la utilización a través de la computadora de múltiples medios como texto, gráficos, sonido, imágenes, animación y simulación combinados y controlados de forma interactiva, para conseguir un efecto determinado.

Las tres formas básicas en que el destinatario recibe la información de un producto de multimedia son: texto, imágenes (estáticas o animadas) y sonido. Las estadísticas indican que los humanos retenemos:

20% de lo que vemos
30% de lo que oímos.
50% de lo que vemos y oímos
80% de lo que vemos, hacemos y oímos

Por lo que se rescata el efecto que brindan los sistemas multimediales ofreciendo una perfecta combinación de componentes.

A modo de ejemplo podemos citar un cd multimedial sobre la Segunda Guerra Mundial. Esta puede presentar la información de una manera más atractiva que un Hipertexto del mismo tema. Podría incluir artículos textuales sobre los países intervinientes, se podrían incluir audio de los sobrevivientes, videos o filmaciones registradas.

2.1.3 Hipermedia

Hipermedia es entonces la combinación del Hipertexto y multimedia. Si la esencia del Hipertexto es la estructura no lineal de la información y la de multimedia la integración de informaciones en distintos formatos, entonces la Hipermedia es un sistema en el que se puede "navegar" en un mar de información textual, gráfica y sonora. Es decir, la Hipermedia es la versión multimedia del Hipertexto. Tiene, por lo tanto, propiedades de ambos, además de una serie de propiedades únicas, que le son propias, y que emergen de esa síntesis.

A continuación listaremos algunas características típicas de la Hipermedia:

- ✓ Se puede utilizar la Hipermedia tanto para leer o escribir información.
- ✓ Su información incluye estructuras no secuenciales que pueden seguirse de diferentes maneras.
- ✓ Las estructuras de información deben seguir asociaciones naturales.
- ✓ La información debe ser estructurada jerárquicamente.
- ✓ Cada unidad de información debe ser presentada en una "pantalla diferente".
- ✓ Debe ser posible compartir parte de la información entre diferentes usuarios.

La Hipermedia nos permite comunicarnos de una manera más efectiva, ya que es relacional y multimedia y, por lo tanto, es más cercana a nuestro modo habitual de expresión y pensamiento.

La Hipermedia es un medio que utiliza y relaciona varias áreas del conocimiento humano tales como Ciencias de la comunicación, Ciencias cognitivas, Ergonomía y factores humanos, Sistemas, Informática, Psicología, y otros.

Algunas definiciones de Hipermedia son:

- "Hipermedia simplemente extiende la noción de texto en el Hipertexto incluyendo información visual, sonora y otros tipos de datos …". [39]

- La *Hiper-media* son presentaciones que se ramifican, en respuesta a las acciones del usuario, o sistemas de palabras o imágenes preorganizadas que pueden explorarse libremente o consultarse a través de estilos específicos. Ellas no serán "programadas" sino diseñadas, escritas, dibujadas y editadas ... Como la prosa y cuadros comunes serán *media* y debido a que en algún modo también son "multi-dimensionales" podemos llamarlas *hiper-media*, siguiendo el uso matemático de la palabra *"hiper-"*... .[39]

- "Hipermedia es una extensión de la idea de *Hipertexto* que incorpora otros componentes tales como: video, ilustraciones, diagramas, voz y animación, así como imágenes generadas por computadora. Generalmente un autor crea las *ligas o links* entre los distintos medios; texto, gráficos, diagramas, fotografías, video, música, películas u otros medios. ..". [28]

- Hipermedia es ″una extensión de Hipertexto, un concepto que designa narrativa altamente interconectada o información vinculada″. [40]

Para clarificar la terminología, caben distinguirse ciertas diferencias entre Hipermedia y multimedia:

Multimedia implica menos interacción con el programa o presentación. Las presentaciones o programas multimediales son secuenciales en términos del flujo de la información.

Los espacios de información en Hipermedia están conectados por links no lineales que el usuario puede seguir en cualquier orden. En cambio los espacios de información en multimedia están organizados secuencialmente con un sólo hilo conductor de la información provista.

El Hipertexto nos provee de una estructura de navegación a través de los datos, mientras que multimedia nos ofrece como punto fundamental una gran riqueza de tipos de datos.

Se pueden mencionar las siguientes ventajas de la Hipermedia:
- mayor facilidad de acceso a datos esparcidos, con menor retardo temporal,
- puede enlazar múltiples datos en red,
- menos espacio físico de almacenamiento,
- puede compartirse por más de un usuario,
- lectura orientada al usuario,
- en potencia podría reunir toda la literatura universal,
- estructura/relacional semántica de los datos orientada al usuario,
- utiliza en una sola estructura datos de diversa índole: texto libre (datos no estructurados), redes semánticas (semiestructurados) y tablas (datos estructurados),
- mínimos requerimientos de destrezas de programación, para construir complejas estructuras.

Y con respecto a las desventajas lo siguiente:
- supone aproximadamente un treinta por ciento de retardo en la lectura,
- menor transportabilidad,
- es necesario cierto aprendizaje de manejo de computadoras,
- no existe una única interfaz estándar,
- posibilidad de estructura en spaghetti,
- no hay una definición central de la estructura de datos, ni un camino sencillo para realizar acciones generales o cálculos específicos sobre los datos. [7]

A modo de ejemplo y para cerrar el concepto de Hipermedia. Si le agregamos a la enciclopedia, fotos o audio, convertiremos dicho Hipertexto en Hipermedia. Si le agregamos al cd multimedial de la Segunda Guerra Mundial la posibilidad de que el usuario sea quien decide que información ver en cada momento, estaríamos convirtiendo la aplicación de multimedia en una aplicación de Hipermedia.

2.2 ¿ Qué es Hipermedia Física?

2.2.1 Definición e Historia

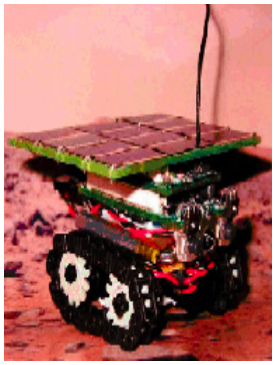
Como ya se comentó en la introducción el primer paradigma que apareció para unir el mundo virtual con el real fue la *Augmented Reality* [33]. En lugar de sumergir a las personas en un mundo virtual artificialmente creado, la meta es aumentar los objetos del mundo físico reforzándolos con una riqueza de información digital y capacidades de comunicación.

La información electrónica a menudo aparece como demasiado desconectada del mundo físico. Por lo general utilizamos los objetos físicos en un ámbito, y la información electrónica relacionada en otro totalmente separado. La *Augmented Reality* se dirige a los problemas de re-integrar la información electrónica con el mundo real. Estos acercamientos tienen como meta en común permitir a las personas que aprovechen las habilidades existentes actuando recíprocamente con los objetos del mundo cotidiano mientras se benefician con el poder de conectarse a una red de computadoras. El ejemplo típico de esto, consiste en lugar de intentar reemplazar los documentos con imitaciones electrónicas de ellos, se crean "documentos interactivos" que se unen directamente con aplicaciones en línea.

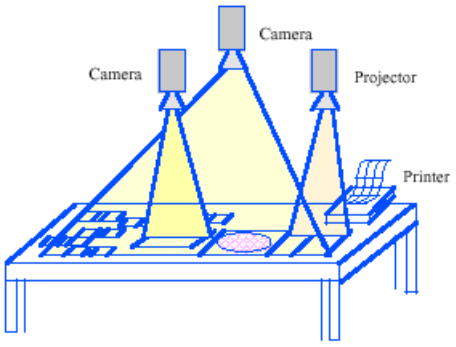
Se puede aumentar la realidad a partir de estas tres estrategias:

- Aumentar al usuario: El usuario lleva un dispositivo, normalmente en las manos o en la cabeza, para obtener información sobre los objetos físicos. Ejemplos de esto pueden ser dispositivos del estilo palms que muestren información o también cascos de realidad virtual.
- Aumentar los objetos físicos: En este caso los objetos físicos cambian según lo que esté delante de ellos. Generalmente implica agregar sistemas de censores o GPS, y un ejemplo de las aplicaciones que se relaciona con esto son aquellas que implican posicionamiento.
- Aumentar el ambiente que rodea al usuario y a los objetos: Ni el usuario, ni el objeto son directamente afectados. En cambio, los dispositivos proporcionan y coleccionan información del ambiente circundante, desplegando la información hacia los objetos y capturando la información sobre las interacciones del usuario con ellos. Ejemplo de estos son cámaras que capturan las actividades que realiza una persona y dependiendo de ello generan cambios en algún dispositivo.

Aumentar al usuario[3]



Aumentar los objetos físicos[16]



Aumentar el ambiente que rodea al usuario rodea al usuario y a los objetos[33]

En la primera figura vemos un ejemplo de un usuario aumentado con una cámara (y otros accesorios) que le permiten acceder a la realidad aumentada. En el caso de la segunda fotografía vemos como se aumenta un objeto móvil con un censor. En la última vemos la representación de un ambiente aumentado con cámaras y proyectores.

La integración de objetos físicos y virtuales no siempre es fácil. La realidad aumentada puede resolver problemas del usuario, pero también puede crear otros. Por ejemplo, supongamos un caso tan simple como un documento. Es evidente que queremos que se registre la información de nuestra interacción con el mismo. Pero ¿Cuál sería la actividad que debería registrarse si decidimos borrar un fragmento del mismo? Marcar con el lápiz el papel es simple y sencillo. Necesitaríamos ir revisando las distintas líneas de texto. Pero al querer realizar una tarea como es la de borrar, es acá donde empiezan a surgir preguntas acerca de esta tarea. Por ejemplo: ¿Si se borra alguna línea qué permanece físicamente en el papel?, o lo que es aún peor, ¿Qué sucede si se borra en el documento digital ? ¿Hay que también actualizar el documento físico? ...Entre otros tantos interrogantes. Como conclusión de este pequeño ejemplo, se desprende que conviene utilizar la realidad aumentada cuando los métodos de los objetos físicos y digitales actúan de forma sincrónica, caso contrario genera demasiadas complicaciones.

Al diseñar las aplicaciones de realidad aumentada, es importante considerar cómo hacer la integración de lo real y lo virtual de la manera más clara posible. Cada aplicación debe elegir la mejor combinación de técnicas para descubrir la información del mundo real y presentarla electrónicamente al usuario. Por ejemplo, varias opciones están disponibles para rastrear la posición del usuario. En cada caso, la opción depende en la naturaleza de la aplicación.

La realidad aumentada (Augmented Reality) es una variación de los ambientes virtuales (Virtual Enviroments), o realidad virtual como se llama más comúnmente. Las tecnologías de ambiente virtual sumergen totalmente al usuario dentro de un ambiente sintético. Mientras que está sumergido, el usuario no puede ver el mundo real que lo rodea. En contraste, la Realidad Aumentada permite que el usuario vea el mundo real, con los objetos virtuales sobrepuestos sobre o compuestos con el mundo verdadero. Por lo tanto suplementa la realidad, en vez de sustituirla completamente. Idealmente, el usuario percibe que los objetos virtuales y reales coexisten en el mismo espacio.

Existen por lo menos seis clases de potenciales aplicaciones de Realidad Aumentada que ya han sido exploradas, a saber: visualización médica, mantenimiento y reparación, anotación, planeamiento de la trayectoria de robots, entretenimiento y aplicaciones militares de aviación[2].

Figura 1:Ejemplo de Realidad Aumentada en aplicaciones de entretenimiento deportivas[3].

Siguiendo en la línea de investigación pero más concretamente si nos referimos a aumentar los objetos físicos resulta interesante referirnos a sistemas que incluyen posicionamiento o ubicación.
En contraste con la búsqueda web tradicional, los sistemas basados en ubicación permiten a los usuarios buscar información en áreas geográficas específicas. Esto se realiza mediante un mapa donde el usuario puede indicar el área de interés y entonces puede ejecutar la búsqueda. Cuando los usuarios se mueven a través del espacio, el sistema debe ordenar y ver la información de su vecindad, y sólo muestra información cuando alcanza un cierto umbral.

Existen también otros sistemas de información basados en ubicación, donde la idea básica es conectar partes de información digital a una coordenada de latitud-longitud específica vía algún

dispositivo móvil. Luego, los usuarios con algún dispositivo móvil, acceden a esta información. De esta manera, los usuarios conseguirán la impresión de que la información digital realmente se halla en cierto modo en ese lugar.

Un ejemplo de esto es GeoNotes[15]. Basado en tecnología de posicionamiento permite añadir notas virtuales a través de algún dispositivo móvil a ubicaciones del mundo real. Cuando otra persona pasa por este sitio le llegará a su dispositivo móvil una notificación, y podrá leerlas. De esta manera una persona podría dejar un graffiti por ejemplo en la puerta de un hotel a través de su dispositivo móvil, y recomendar o no ciertas características del mismo. Y otra persona al pasar frente a él por ejemplo podría reforzar su elección de hospedarse allí basada en las notas que lea desde su PDA ó dispositivo móvil.

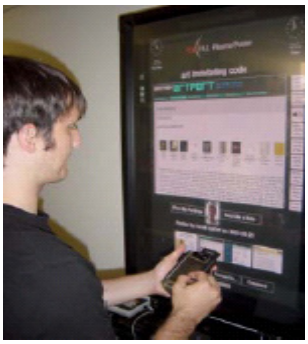


Figura 2:Ejemplo de servicio para anotaciones a través de una PDA[8].

Es interesantísimo el espectro de aplicaciones que pueden incluir este tipo de sistemas. El ejemplo antes mencionado es meramente social. Otro ámbito de utilización de los mismos podría ser museos, organizaciones turísticas o centros culturales. En estos ejemplos de manera contraria a lo que sucede con los graffitis la información provista suele ser profesionalmente creada y tiende a ser seria y 'útilmente orientada'. Es decir se aumentan los objetos reales con información digital que le llegará al dispositivo de la persona que se encuentra frente a él. La desventaja de estos sistemas es que por lo general la información se encuentra muy estructurada y en muchas ocasiones no se ajustan a las expectativas de los usuarios.

El sistema de GeoNotes tuvo como propósito unir áreas bastante dispares de la investigación dentro de la comunidad de Interacción del ser Humano con la Computadora. Cabe destacar que intentó disminuir el límite entre el espacio físico y el digital, y al mismo tiempo las disputas para reforzar el espacio digital socialmente. Se permite a los usuarios dejar rastros virtuales que se "añaden" a las posiciones geográficas específicas. De esta manera se enriquece el conocimiento social que ya existe en el espacio físico, cuando estos rastros están disponibles para otros usuarios.

En este punto, que ya hemos obtenido algo de terminología, es necesario hacer una distinción: La *Augmented Reality* es un subconjunto de un tipo de aplicaciones conocidas como *Mixed Reality*.

La M*ixed Reality*[38] fue definida por Paul Milgram como la "combinación de mundos reales y virtuales en alguna parte a lo largo de la ' serie continua de la virtualidad ' que conecta ambientes totalmente verdaderos con los totalmente virtuales."

Dentro de esta definición se suelen integrar los siguientes términos, Mixed Reality Continuum, Augmented Reality, Augmented Virtuality, Ubiquitous Computing, Wearable Computing entre otros.

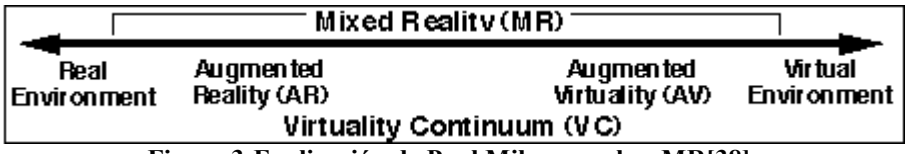


Figura 3:Explicación de Paul Milgram sobre MR[38].

Como bien explica la Figura 3, Mixed Reality puede definirse como el conjunto de varias subclases de tecnologías que van desde el ambiente Real al Virtual.

Debido a las recientes tendencias hacia la movilidad y a la utilización de redes de computadoras, se han aumentado notoriamente las experiencias interactivas donde los dispositivos móviles aumentan al mundo real. De a poco se comienza a transitar el camino hacia la inmersión de este nuevo conjunto de tecnologías que incluye la *Mixed Reality*. No cabe duda de que la Hipermedia juega un rol central en cualquier tipo de aplicación interactiva. Por lo que resulta natural pensar que la misma Hipermedia evoluciona con estos nuevos tipos de aplicaciones que comienzan a surgir. Es evidente que ciertos mecanismos permanecen presentes pero la inclusión del mundo real en el mundo digital despierta la necesidad de nuevas maneras de estructurar la información, ya sean modelos o paradigmas.

La Hipermedia se basa y se implementa bajo el paradigma orientado a objetos, permitiendo construir aplicaciones de *Mixed Reality*. Hipermedia juega un papel central en este tipo de aplicación como una manera de integrar los diferentes medios de comunicación y estructurar la información que se manipula por las aplicaciones. También es una manera natural de guardar las asociaciones entre los diferentes materiales en una exploración del mundo real.

Existen muchas actividades que se caracterizan por el hecho de que las personas tienen que tratar con información que existe en una mezcla de medios de comunicación físicos y digitales. La administración pública, hospitales, y fabricación son ejemplos de dominios dónde existen mezclas de materiales digitales y físicos en las rutinas de trabajo diarias. Estos dominios desafían a los sistemas de Hipermedia Tradicional puramente digitales, y a los otros tales como los sistemas de “open hypermedia”, “spatial hypermedia”, y la web. Se necesitan desarrollar técnicas que incorporen los objetos del mundo físico en los sistemas de Hipermedia; esto se denomina en [26] como "physical hypermedia”.

Cuando se desea desarrollar Hipermedia como apoyo para crear y mantener las relaciones entre los materiales físicos y digitales, se necesitan analizar los diferentes tipos de relaciones que tendrían sentido en el mundo mixto. Pudiendo ir entre objetos puramente físicos a completamente digitales.

A modo de ejemplo podemos citar el sistema Topos que se desarrolló en el proyecto de WorkSPACE. Su objetivo fue desarrollar un sistema de Hipermedia Física para dar a los usuarios, un ambiente familiar al mundo real en el que trabajan. Organizando las mezclas entre materiales digitales y físicos, mediante colecciones de objetos. El concepto central en Topos es el *workspace* que es el medio principal para agrupar y organizar los materiales y objetos en el espacio 3D. Otro aspecto interesante que provee es que permite agrupar, mezclar y conectar los espacios de trabajo de varias maneras diferentes.

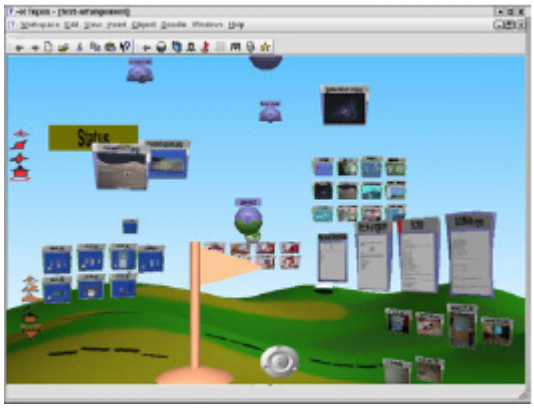


Figura 4: Un workspace de Topos con un terreno como fondo Y documentos organizados en varios grupos espaciales[24].

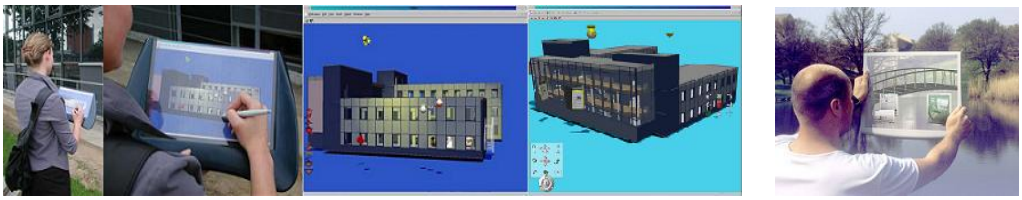


Figura 5: Ejemplo del Proyecto Topos de aumentar la realidad con información digital en proyectos arquitectónicos[5].

Otro proyecto interesante sobre Hipermedia Física es el framework HyCon[26] para sistemas de Hipermedia context-aware. Su arquitectura da soporte para realizar anotaciones, manejo de links y tours guiados asociando ubicaciones y objetos con dispositivos RFID o Bluetooth con mapas, páginas web y colecciones de recursos. Introduce el uso de XLinks y un nuevo sistema de búsqueda para la web llamado Geo-Based Search (GBS). Lo interesante de este sistema es que soporta los mecanismos clásicos de Hipermedia para navegar, buscar, hacer notaciones y tours guiados en el mundo físico, permitiendo a los usuarios realmente unir objetos digitales y/o físicos. Cabe destacar también que esta arquitectura propuesta, incluye interfaces para incluir la capa de censores, que encapsulan el GPS y otros sistemas de posicionamiento que se pueden utilizar.

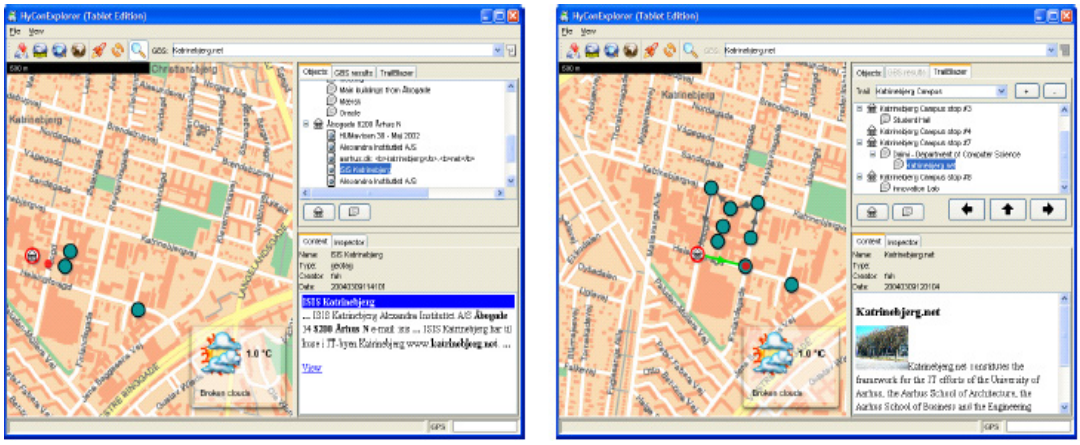


Figura 6: Ejemplo de navegación y búsqueda en el prototipo HyConExplorer[26].

La figura 6 muestra una manera de navegar y buscar información en el prototipo. Para ello se ingresan ciertos términos, y el sistema captura la información de contexto. La ubicación actual del usuario se muestra con un punto rojo, las anotaciones con círculos verdes y el lugar elegido para visitar se encuentra remarcado. También se puede observar información digital de la misma, y ciertos tours recomendados en forma de flechas.

Si hablamos de Hipermedia que ejemplo más oportuno que la web, tan profundamente arraigada a las costumbres propias de la vida cotidiana. Es interesante estudiar como el ambiente local, se ignora completamente con respecto a la información dinámica que se ofrece. En el enfoque web la información oculta el hecho que el mundo también está compuesto de artefactos físicos. Por consiguiente, se piensa que la próxima dirección para la web es unir lo físico y lo virtual [27]. No resultaría tan raro pretender que la web nos diera una visión más cercana a la realidad que la actual. Dicho en otras palabras sería más mucho más práctico percibir que la web se uniese a la realidad, que pretender que la realidad se adapte a la web. Una de las metas que se están estudiando actualmente consiste en aumentar la realidad dando al Hipertexto, y a la Hipermedia, una presencia física en el mundo real. El proyecto proXimity es un sistema que busca aumentar la realidad dando al Hipertexto, y así a la Web, una presencia física en el mundo verdadero. Está basado en trabajos sobre Hipermedia y movilidad en el mundo real e intenta extender la metáfora del link de Hipermedia en el mundo real.

Por ejemplo este proyecto podría aplicarse a escenarios de viajes y ubicar objetos físicos en el mundo real. Estos serían accedidos por medio de links en el ambiente virtual. Es decir, por ejemplo una persona que se encuentra de paseo por una ciudad que no conoce, podría ir obteniendo información en su dispositivo móvil que lo ayude a orientarse e informarse. Aquí surge el concepto de " walk the Links" [27] o *"caminar los links"*.

Podrían usarse las técnicas de Hipermedia para aumentar el mundo real así como se han usado las técnicas del mundo real para aumentar el Hipermedia. Para citar un ejemplo, si se encuentra visitando Francia, y está frente a la Torre Eiffel se podría mostrar en el dispositivo móvil información sobre sus características, o cualquier otra información del sitio web oficial de la misma. También como ya se introdujo con el GeoNotes, se podrían incluir anotaciones de otros visitantes. Lo interesante que se podría agregar a esto, es la proliferación de los links al mundo real; y de esta manera los usuarios podrían "caminar los links" entre instancias del mundo real usando información del mundo virtual. Siguiendo el ejemplo varios lugares turísticos de Francia podrían estar conectados por "links caminables" y el usuario frente a la Torre Eiffel podría ser guiado hasta el Arco del Triunfo de la Estrella a través de un link caminable. Al elegir este link podría mostrarse en el dispositivo un mapa interactivo, una foto, o una descripción del camino a recorrer. Notar aquí que la persona podría elegir o no *Caminar el link,* mantener su ubicación, o seguir su itinerario planeado.

El principal aporte de proXimity es establecer tres componentes para el Hipertexto en la Web. A saber un componente espacial, uno temporal y otro semántico. El componente semántico permitirá agrupar los links bajo algún criterio. El temporal dará la información de la ubicación tanto de la persona como de los objetos del mundo real, y el espacial resolverá las cuestiones de como atravesar las distancias. Este trabajo establece fuertes indicios para suponer que en el futuro, el Hipertexto y el mundo real van a unirse y permitirán que el usuario en efecto "camine los links".

Como ya se estudió en [22] el modo de navegación difiere entre la Hipermedia tradicional y la Hipermedia Física. Si nos referimos al momento de activación, queda claro que en la Hipermedia Tradicional esto ocurre al navegar un link que nos lleva a ese nodo. En cambio en la Hipermedia Física esto ocurre de manera implícita cuando el usuario se encuentra parado frente a un objeto físico determinado. Es decir por algún mecanismo de detección de ubicación, se determina que el usuario está parado frente a un "objeto aumentado" y se le proporciona cierta información. Aquí dependiendo de la naturaleza de la aplicación deben analizarse algunas otras cuestiones. Por ejemplo como será el mecanismo de feedback para la notificación de la llegada del usuario a un nuevo objeto aumentado, que información proveerle en ese momento, si se mantiene o no información de contexto sobre los lugares ya visitados, etc.

Siguiendo esta última idea es importante aclarar que en los links de Hipermedia Tradicional, el link contiene información que especifica el objeto fuente, que a menudo es, por ejemplo una URI. Cuando se permite a los links tener "anclas físicas" (es decir, que lo que denota al link es un objeto físico), su resolución puede volverse mucho más compleja. Por ejemplo: podrían usarse sistemas de ubicación para descubrir una persona frente a un "ancla física", o una interfaz explícita de activación (tal como un lector de barras) que sea activada en la localización física del ancla (al ser explorada); o el ambiente del usuario podría ser implícitamente supervisado y luego analizando dicha información comparar contra marcadores simbólicos.

Dentro de los ambientes físicos las anclas físicas deben indicarse vía alguna forma de señal. Es decir, el usuario tendría que poder percibir que está frente a un objeto que puede brindarle más información de lo que simplemente está observando. Las señales podrían ser direccionadas digitalmente, como por ejemplo con pantallas para la representación visual o altavoces para representación sonora que informen al usuario. Una estrategia alternativa es proporcionar una representación digital del mundo físico (tal como un mapa), en el cual las anclas físicas pueden ser colocadas de forma digital. También se podría usar Realidad Virtual, o Realidad Aumentada. Uno de los problemas que aparecen aquí es que la notificación de la presencia de un ancla desvía el foco de la tarea de un usuario. Una modalidad orientada a la pregunta-explícita aparece así más atractiva que

la exhibición universal, a menos que las anclas estén garantizadas por algún criterio como podría ser personalización de una determinada aplicación.

Cuando un link existe en el mundo digital, la idea de trasversal, es a menudo bastante transparente para usuario, pero en el mundo físico, la carga puede estar en el usuario que tiene que realizar lo trasversal. Por ejemplo, se podría tener un link en un monumento histórico a otro en la misma ciudad: una vez que el sistema tiene resuelto el ancla de la fuente, para seguir el link el usuario debe moverse al destino.

Queda claro entonces, que en las aplicaciones de Hipermedia Física los "objetos aumentados" deben caracterizarse bajo algún criterio que permita individualizarlos. Será este el que permita a la aplicación detectar cuando una persona se encuentra delante de ellos. Como se propone en [22] desde el paradigma orientado a objetos existen dos cuestiones fundamentales a la hora de pensar en el diseño de estos objetos físicos. Más precisamente, deberá hacerse hincapié en el comportamiento esperado de la aplicación cuando el usuario se posiciona frente a un objeto o cuando desea obtener información de cómo llegar a este desde la posición actual.

2.2.2 Resumen de Hipermedia Física

Resumiendo el paradigma de Hipermedia Física (HF) permite construir relaciones significativas entre el mundo real y digital usando el paradigma de Hipermedia. El término Hipermedia Física ha sido utilizado para describir cualquier sistema de Hipermedia que trata con las propiedades del mundo físico.

Podemos entender la Hipermedia Física como un conjunto de nodos, virtuales y físicos interconectados. Es decir un objeto físico se aumenta con algún tipo de información, que se le presenta a un usuario al estar frente a él en su dispositivo móvil. Esta información puede ser de cualquier naturaleza, y lo interesante de esto es que puede incluir links a otros objetos físicos que en este caso deberían ser "caminables por el propio usuario en el mundo real".

La movilidad ciertamente parece imponer nuevas restricciones en el uso de tecnología de información. La habilidad de acceder a la Web desde los dispositivos móviles se ha vuelto trivial con los Browser WAP encontrados en muchos teléfonos móviles y PDAs. Cosas que los usuarios no querían hacer, no se atrevían a hacer, o no tenían tiempo, las personas querrán hacerlas con sus dispositivos móviles en cafés, en el transporte público, o los descansos escolares.

Capítulo 3 Extensión del MVC para soportar Hipermedia Física

3.1 Arquitectura del MVC y sus 3 componentes

3.1.1 Arquitectura

Para lograr una mejor comprensión de los fundamentos del MVC nos parece apropiado citar sus orígenes y como surge a partir de arquitecturas acopladas. En estas el programa era dividido a menudo en dos partes: El "Modelo", que representaba el funcionamiento interior del programa dónde el procesamiento realmente tenía lugar. Y la "Vista" que representaba la entrada y salida del programa que interactuaba con el usuario. La vista tomaba la entrada del usuario y pasaba la información hacia el modelo. La información de salida del modelo era tomada por la vista y luego se la presentaba al usuario.

Es evidente que en este modelo ambas capas están profundamente ligadas. Para lograr mejores resultados se necesita que el modelo opere independientemente de la manera en que la vista actúa con el usuario y que la vista satisfaga las necesidades de la interacción, independientemente del modelo que funcione debajo. A continuación citamos algunas de las consideraciones necesarias para ello:

1. Las entradas y salidas del modelo no necesariamente se corresponden con las entradas y salidas de las vistas.
2. Un modelo puede trabajar con varias vistas.
3. Una vista puede trabajar con varios modelos.
4. En general, para que el modelo y la vista se puedan comunicar se necesita un "adaptador" para traducir la entrada de uno al otro y viceversa.
5. Ni el modelo ni la vista son capaces de crear el adaptador que los conecta porque cada uno es independiente del otro.

Estas y otras tantas razones dejaron al descubierto la necesidad de agregar un componente más, el cual se denominó "Controlador". Este hace las veces de intermediario entre ambas capas, "sabe" sobre el modelo y la vista que sé están utilizando. Algunas de las responsabilidades que se le atribuyen son:

- Instanciar el modelo y las vistas.
- Instanciar el adaptador o adaptadores que permitirán la comunicación entre el modelo y las vistas.
- Establecer las conexiones entre los adaptadores y las vistas y entre un adaptador y el modelo.

La inclusión de este tercer elemento (el controlador), separa el modelo de las vistas, permitiendo perder acoplamiento y tener la posibilidad de que cada uno cambie de manera independiente. Esta arquitectura resultante se conoce como MVC(*Model View Controller*). Este patrón de diseño consiste en tres componentes. A saber: el Modelo que representa al domino, la Vista que es la presentación que se ofrece al usuario, y el Controlador que define la manera en que la interfaz de usuario reacciona a la entrada de la persona. El principal aporte del MVC es permitir desacoplar las capas para aumentar la flexibilidad y re-uso.

Veamos las diferencias del modelo MVC con los modelos convencionales. En el esquema más básico de programa, se tiene una entrada o parámetros que llegan (conocido más comúnmente como INPUT), estos se procesan y se muestra el resultado (generalmente denominado OUTPUT). El siguiente diagrama muestra esta situación:

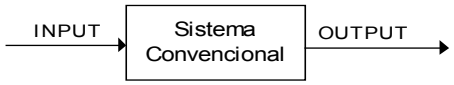


Figura 7:Esquema de programa tradicional [41].

En el caso del patrón MVC el procesamiento se lleva a cabo entre sus tres componentes. El controlador recibe una orden y decide quien la lleva a cabo en el modelo. Una vez que el modelo (la lógica de negocio) termina sus operaciones devuelve el flujo, que vuelve al controlador y este envía el resultado a la capa de presentación. Esto se ve en el siguiente diagrama:

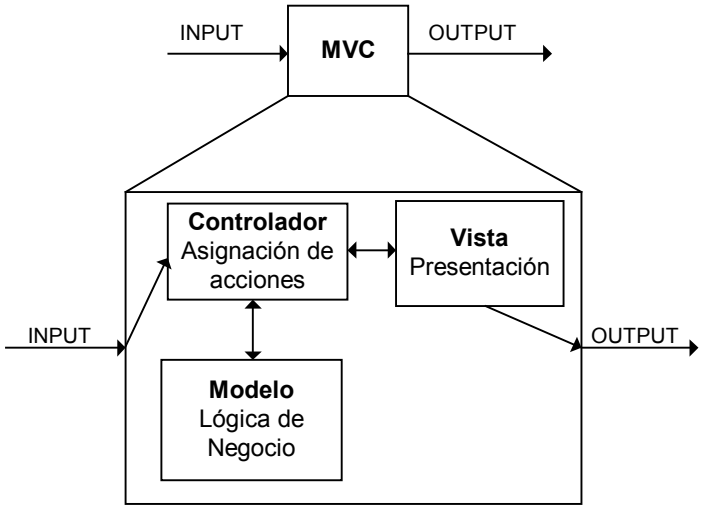


Figura 8:Esquema de programa con la visión del MVC[41].

La arquitectura MVC (*Model /View /Controller*) fue introducida inicialmente en la comunidad de desarrolladores de Smalltalk-80. Fue diseñada para reducir el esfuerzo en los desarrollos de sistemas múltiples y sincronismo de datos. El patrón MVC original de Smalltalk-80 se muestra en el siguiente diagrama:

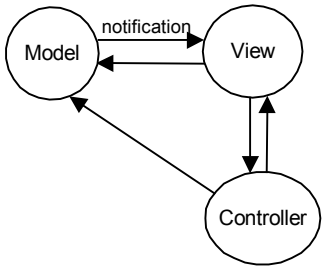


Figura 9: Diagrama del MVC propuesto para Smalltalk-80 [29].

La principal característica de esta arquitectura es que los Modelos, las Vistas y los Controladores se tratan como entidades separadas. Esta clara separación entre los componentes de un programa, permite implementarlos por separado y de forma independiente entre los mismos. Otra característica es que la conexión entre los Modelos y sus Vistas es dinámica, o sea, se produce en tiempo de ejecución, no en tiempo de compilación. Por lo tanto el modelo no está unido a una vista en particular, estas pueden ir variando según la ejecución del programa. Cualquier cambio producido en un Modelo se reflejará automáticamente en cada una de las Vistas del mismo.

Por lo tanto, al incorporar el modelo de arquitectura MVC a un diseño, las partes de un programa se pueden construir por separado y luego unirlas en tiempo de ejecución. Si uno de los Componentes funciona mal, puede reemplazarse sin que las otras partes se vean afectadas.

La arquitectura MVC en su forma más general tiene un Modelo, múltiples Controladores que manipulan ese Modelo, y varias Vistas de los datos del mismo, que cambian según el estado del Modelo.

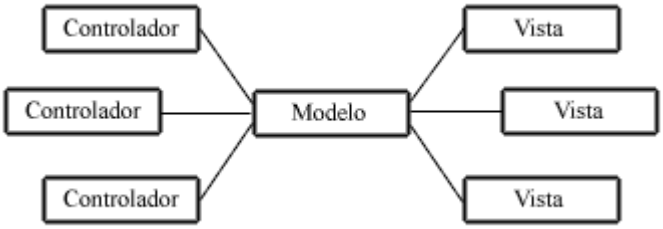


Figura 10:Esquema del MVC en su forma más general [45].

3.1.2 Definición de cada uno de los componentes del MVC

El *Modelo* representa los datos del programa. Maneja los datos y controla todas sus transformaciones. El Modelo no tiene conocimiento específico de los Controladores ni de las Vistas, ni siquiera contiene referencias a ellos. Es el propio sistema el que tiene encomendada la responsabilidad de mantener enlaces entre el Modelo y sus Vistas, y notificar a las Vistas cuando cambia el Modelo.

La *Vista* maneja la presentación visual de los datos representados por el Modelo. Genera una representación visual del Modelo y muestra los datos al usuario.

El *Controlador* proporciona significado a las órdenes del usuario, actuando sobre los datos representados por el Modelo. Cuando se realiza algún cambio en la información del Modelo o se altera la vista entra en acción. Interactúa con el Modelo a través de una referencia al mismo.

En conclusión el MVC es un patrón ampliamente utilizado en múltiples plataformas y lenguajes. Algunos de los principales beneficios de este patrón son:

- Menor acoplamiento:
  - Desacopla las vistas de los modelos.
  - Desacopla los modelos de la forma en que se muestran e ingresan los datos.
- Mayor cohesión:
  - Cada elemento del patrón esta altamente especializado en su tarea (la vista en mostrar datos al usuario, el controlador en las entradas y el modelo en su objetivo de negocio).
- Las vistas proveen mayor flexibilidad y agilidad:
  - Se pueden crear múltiples vistas de un modelo.
  - Se pueden crear, añadir, modificar y eliminar nuevas vistas dinámicamente.
  - Las vistas pueden anidarse.
  - Se puede cambiar el modo en que una vista responde al usuario sin cambiar su representación visual.
  - Se pueden sincronizar las vistas.
  - Las vistas pueden concentrarse en diferentes aspectos del modelo.
- Mayor facilidad para el desarrollo de clientes ricos en múltiples dispositivos y canales:
  - Una vista para cada dispositivo que puede varias según sus capacidades.
  - Una vista para la Web y otra para aplicaciones de escritorio.
- Más claridad de diseño.
- Facilita el mantenimiento.
- Mayor escalabilidad.

3.2 Descripción de la funcionalidad

El MVC separa el modelo de la vista estableciendo un protocolo suscripción/notificación entre ellos. La vista debe asegurar que su apariencia refleja el estado del modelo. Siempre que los datos del modelo cambian, el modelo notifica a las vistas que dependen de él. En la contestación, cada vista consigue una oportunidad de actualizarse. Este acercamiento permite unir múltiples vistas a un modelo proporcionado diferentes presentaciones. Se pueden crear nuevas vistas para un modelo sin rescribirlo.

Smalltalk-80 desarrolló la idea de objetos dependientes, para manejar las notificaciones de cambio. Las vistas y controladores de un modelo son registrados en una lista como dependientes de ese modelo, para ser informados siempre que algún aspecto del modelo cambie. Cuando un modelo cambia, se transmite un mensaje para notificar a todos sus dependientes sobre el cambio. Este mensaje puede ser parametrizado, porque puede haber muchos tipos de mensajes de cambio. Cada vista o controlador responde a los cambios del modelo de la manera adecuada.

Aunque se pueden encontrar diferentes implementaciones de MVC, el flujo de control generalmente es el siguiente:

- El usuario interactúa con la interfaz de alguna manera (Ej. Presionando un botón, enlace).
- El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario.
- El controlador accede al modelo, posiblemente actualizando los datos enviados por el usuario.
- El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario.
- La vista usa el modelo para generar la interfaz apropiada para el usuario donde se reflejan los cambios en el modelo. En algunas implementaciones la vista no tiene acceso directo al modelo, dejando que el controlador envíe los datos del modelo a la vista.
- La interfaz espera por nuevas interacciones del usuario para iniciar nuevamente el ciclo.

Si analizamos bien, los componentes de esta arquitectura, y la manera en que se relacionan, descubriremos que aparecen también otros patrones de diseño incluidos en el MVC. Por ejemplo, los dependientes de un objeto son informados de sus cambios, sin que el objeto que cambió conozca la existencia de los mismos. Es aquí donde se aplica el patrón de diseño Observer[19]. O sea, se utiliza este patrón para el mecanismo de publicación y suscripción que permite la notificación de los cambios en el modelo a las vistas.

También se podría señalar la relación con el patrón de diseño Composite[19]. Utilizando este patrón se podría crear una jerarquía de vistas y tratar a cada vista compuesta igual que a una vista normal. Por ejemplo se podría pensar en un inspector de objetos compuestos, donde la vista del objeto es en realidad el conjunto de todas las vistas que lo componen. Dicho en otras palabras hay una vista que contiene vistas más específicas.

El MVC también permite cambiar la manera que una vista responde a la entrada del usuario sin cambiar la presentación visual. El MVC encapsula el mecanismo de contestación en un objeto Controlador. Por lo tanto para lograr diferentes efectos podría crearse una jerarquía de controladores, que permitan construir un nuevo controlador como una variación de uno ya existente. Una vista usa una subclase de controlador para llevar a cabo una estrategia de contestación en particular; para realizar una estrategia diferente, simplemente se reemplaza con un tipo diferente de controlador. La relación de Vista-Controlador es un ejemplo del patrón de diseño Strategy[19]. Utilizando este patrón se puede cambiar dinámicamente o en tiempo de compilación los algoritmos del controlador mediante los cuales responde a su entorno.

El MVC contiene principalmente los patrones de diseño antes mencionados (Observer, Composite, y Strategy)[19]. Además, usa otros patrones de diseño como el Factory Method[19] para especificar la clase del controlador predefinida para una vista y el Decorator[19] para agregar capacidades adicionales a una vista por ejemplo el scrolling. También se relaciona con el Proxy[19], para distribuir la arquitectura (Modelo-Vista-Controlador).

3.3 Componentes agregados

La manera más apropiada que encontramos para encarar la extensión del MVC fue analizar por separado cada uno de los componentes principales e intentar descubrir que aspectos faltaban cubrir para soportar aplicaciones de Hipermedia Física con ubicación.

A partir de analizar cada uno de los componentes, elegimos como estrategia de solución, extender únicamente la parte del controlador. Tomamos esta decisión al ver que la vista y el modelo podían soportar aplicaciones de Hipermedia Física con ubicación, sin necesidad de modificar o agregar elementos.

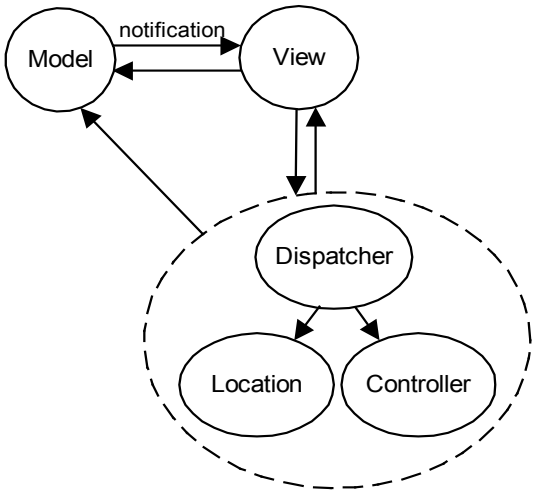
Más precisamente si nos referimos a la Vista, esta necesita mostrar información física además de la digital. Aquí se podrían analizar cuestiones de visualización y diseño de interfaces de usuario, pero queda fuera del alcance de nuestra tesis. Simplemente optamos por utilizar el componente tradicional del MVC, que reúna visualización de información digital y física

El área del Modelo también tendrá información digital y física, pero para esto, tampoco creemos necesario agregar nuevos componentes. En esta tesis consideraremos tanto a los datos físicos como a los digitales como un mismo modelo para la aplicación. Esto no descarta que otros trabajos hagan un análisis más exhaustivo del modelo y descubran una mejor opción de diseño.

Con respecto al Controlador, además de ser el nexo entre el modelo y la vista, tendrá que identificar si se trata de un requerimiento que implica ubicación o no y será el encargado de procesar los pedidos que requieran ubicación. Por lo tanto creemos necesario considerar la incorporación de nuevos componentes que colaboren para resolver dichas funciones.

Concretamente se pueden distinguir dos nuevas acciones que necesitan ser resueltas por el controlador. La primera es detectar la naturaleza del requerimiento. Si esta implica ubicación, entra en juego la segunda, que es determinar unívocamente el objeto frente al cual la persona se encuentra parada. Resumiendo necesitaríamos por lo menos dos nuevos elementos, uno para cada una de las funciones.

A partir del diagrama del patrón MVC, podrían agregarse estos dos elementos antes mencionados, quedando de la siguiente manera:



El Dispatcher será el encargado de ver si es una demanda con ubicación o no. En el caso de serlo pasaría el control al Location, en caso contrario al Controller.

El Location se encargará de buscar la información necesaria para determinar exactamente el pedido. Con esto nos referimos a que es responsabilidad de este elemento precisar cual es el objeto físico que determinó el requerimiento.

El Controller, por su parte seguirá teniendo el funcionamiento tradicional que tenía.

Se puede ver en el diagrama anterior que se mantiene el concepto de MVC. Se puede apreciar también que el controlador se ha expandido en tres elementos, lo que resulta en un controlador más desarrollado. De esta manera se tiene una especialización del MVC que soporta ubicación.

Capítulo 4 Desarrollo del framework que soporte Hipermedia Física

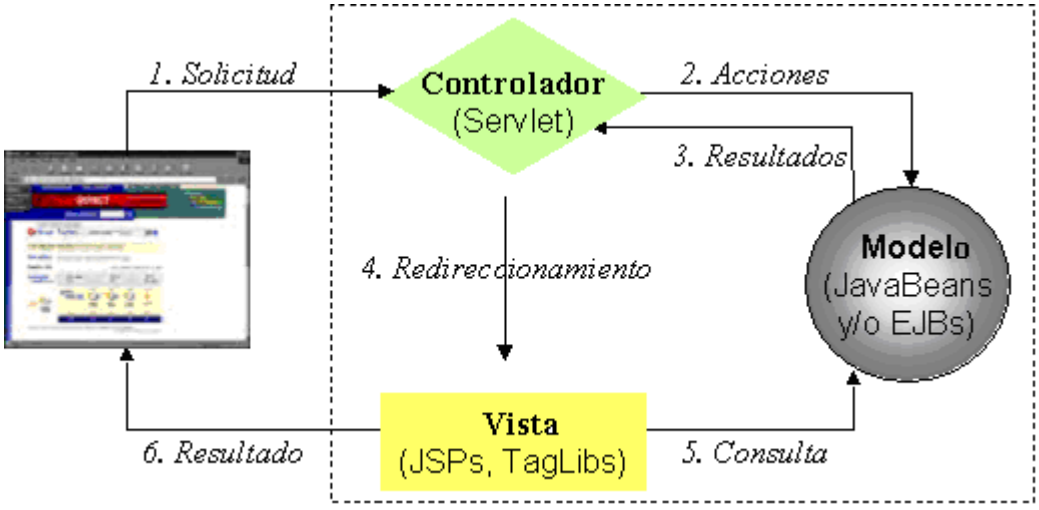
4.1 Struts: Implementación del MVC en la Web

Struts es un framework que implementa el patrón de arquitectura MVC en Java. Es flexible ya que basa su control en tecnología Standard, como son *Java Servlets*, *JavaBeans*, *ResourceBundles*, y *Extensible Markup Language* (XML), etc.

El framework de Struts ayuda a crear ambientes de desarrollo extensibles para las aplicaciones, para esto se basa en las publicaciones standard y patrones de diseño. Struts es un subproyecto del proyecto Jakarta de Apache Software Foundation.

Struts permite que el desarrollador se concentre en el diseño de aplicaciones complejas como una serie simple de componentes del Modelo y de la vista intercomunicados por un control centralizado. Este diseño garantiza la construcción de aplicaciones más consistentes y más fáciles de mantener.

Veamos como funciona Struts en aplicaciones Web:



El navegador genera una solicitud que es atendida por el Controlador (un Servlet especializado). El mismo se encarga de analizar la solicitud, seguir la configuración que se le ha programado en su XML y llamar a la Acción correspondiente pasándole los parámetros enviados. La Acción instanciará y/o utilizará los objetos de negocio para concretar la tarea. Según el resultado que retorne la Acción, el Controlador derivará la generación de interfaz a una o más JSPs, las cuales podrán consultar los objetos del Modelo a fines de realizar su tarea.

El hecho de que Struts se base en el patrón MVC otorga varios beneficios:

- Se puede distribuir el esfuerzo de desarrollo, tal que las modificaciones de implementación en una parte de la aplicación Web no demanden cambios en otros.
- Gracias a la separación de los tres componentes se pueden prototipar más fácilmente los trabajos.
- Se pueden migrar aplicaciones de manera más sencilla, debido a que la vista está separada del modelo y del control y puede ser tolerable a diferentes plataformas y categorías de usuarios.
- El diseño MVC tiene una estructura organizada que mejora la escalabilidad de soporte, permitiendo crear aplicaciones de mayor envergadura, que sean fáciles de modificar y mantener gracias a la clara separación de tareas.

Además de los beneficios antes mencionados, queremos destacar que al estar basado en la arquitectura MVC, se puede utilizar para aplicaciones que requieran diferentes tipos de Browsers.

Esto es posible gracias al concepto MVC que permite tener diferentes vistas, es decir, los datos pueden ser visualizados de diferentes formas. Esto nos proporciona la opción de tener diferentes vistas adaptadas a los diferentes Browser. Esto se puede observar en el siguiente dibujo:

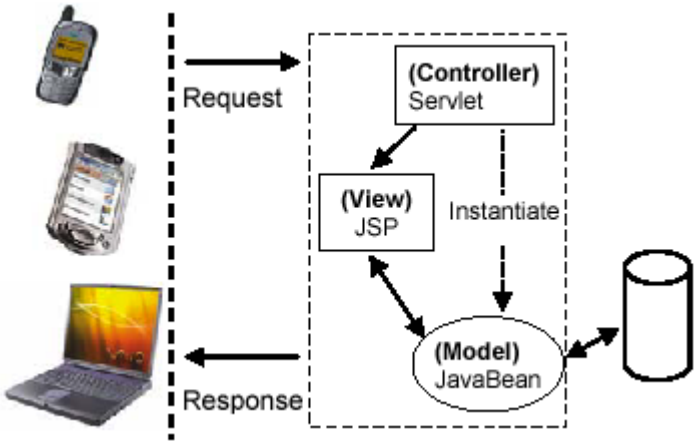


Figura 11: Aplicaciones con diferentes tipos de Browsers [12].

Para llevar a cabo este funcionamiento se deberán tener JSPs (u otras presentaciones) adaptadas a los diferentes Browsers. Por ejemplo si es un Browser Wap, la JSP contendrá código en el lenguaje WML; en el caso de tratarse de un Browser Web la JSP estará escrita en el lenguaje HTML.

Para entender mejor esto, veamos como es la estructura de los componentes comparando dos tipos de aplicaciones: una que basa su interacción en un Browser Web y otra a través de un Browser Wap.

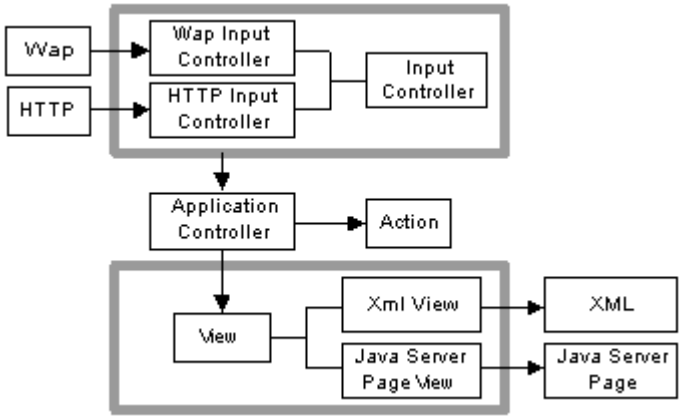


Figura 12: MVC en aplicaciones web [29].

Algunas de las características más significativas de Struts son:

- Configuración del control centralizada.
- Especificación de las Interrelaciones entre Acciones y páginas u otras acciones a través de archivos XML en lugar de codificarlas en los programas o páginas.
- El mecanismo para compartir información bidireccionalmente entre el usuario de la aplicación y las acciones del modelo son los componentes de aplicación.
- Incluye librerías para facilitar la mayoría de las operaciones que generalmente realizan las páginas JSP.

Contiene herramientas para validación de formularios. Bajo varios esquemas que van desde validaciones locales en la página (a través de código en javaScript) hasta las validaciones de fondo hechas al nivel de las acciones.

A continuación describiremos más detalladamente el funcionamiento del "controlador" de Struts y los componentes que lo integran, para brindar una visión completa del mismo que permita comprender la extensión que proponemos más adelante.

El framework de Struts provee varios componentes que constituyen la parte del controlador en el MVC. Esto incluye un servlet de control, manejadores de requerimientos, y varios otros objetos para mantener el control.

El controlador es la parte de la aplicación que recibe los requerimientos del usuario, y decide que funcionalidad del modelo debe ser ejecutada, luego delega la responsabilidad para producir la próxima interfaz del usuario en el componente de la vista correspondiente.

En Struts el componente principal del controlador es un servlet de la clase *ActionServlet*. Este servlet es configurado para definir el conjunto de *ActionMappings*. Un *ActionMapping* define un path que macheará con una URL del requerimiento entrante, y normalmente especifica el nombre de una subclase de la clase *Action*. Los *Actions* encapsulan la lógica de negocios, interpretan la salida, y por último dan el control al componente de la vista para crear la respuesta. Struts también soporta la habilidad de usar clases *ActionMapping* que tienen propiedades adicionales más allá de las estándares con las que opera el framework. Esto permite almacenar información adicional especifica de cada aplicación.

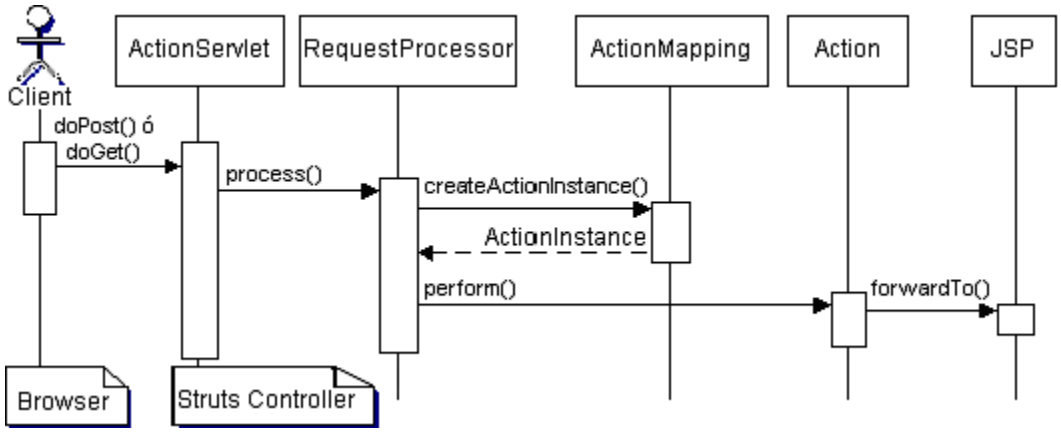
4.1.1El comportamiento del controlador

El controlador parsea los archivos de configuración (principalmente el struts-config.xml) y utiliza esta información para instanciar objetos de la parte del controlador. Estos objetos en forma conjunta forman la configuración de Struts. Esta configuración define los *ActionMappings* para la aplicación, entre otras cosas.

El servlet controlador (*ActionServlet*) utiliza un *RequestProcessor* que consulta a los *ActionMappings* para dirigir los requerimientos a otros componentes del framework. El requerimiento debe ser forwardeado a una JavaServer Page (JSP) o a una subclase de *Action* provista en el desarrollo de la aplicación. Generalmente un requerimiento es forwardeado primero a un *Action* y luego a una JSP (u otro tipo de presentación).

El mapping entre la URL entrante y el path configurado ayuda al controlador a convertir el requerimiento en un *Action* de la aplicación. Un *ActionMapping* puede tener como propiedades: El path del requerimiento (o "URI"), la especificación de una subclase de *Action* para resolver la demanda y otros elementos que le sean necesarios.

Para clarificar, la colaboración de los objetos nombrados hasta el momento, se ofrece el siguiente diagrama:



El *Action* puede manejar el requerimiento y responder al cliente o indicar a quien se le debe dar el control. Por ejemplo, si una persona intenta loguearse, un *LoginAction* puede remitir el requerimiento a una página de menú principal.

El trabajo del controlador es el siguiente:

- Procesar el requerimiento del usuario.
- Determinar lo que el usuario está intentando llevar a cabo según el requerimiento.
- Manipular información del modelo si es necesario para luego ser mostrada en la vista resultante.
- Seleccionar la vista apropiada para responder al usuario.

El *ActionServlet* además de ser el controlador de la aplicación, es responsable de la inicialización y limpieza de los recursos. Al inicializarse el *ActionServlet*, carga la configuración de la aplicación. Para llevar a cabo esta configuración se basa en el archivo web.xml.

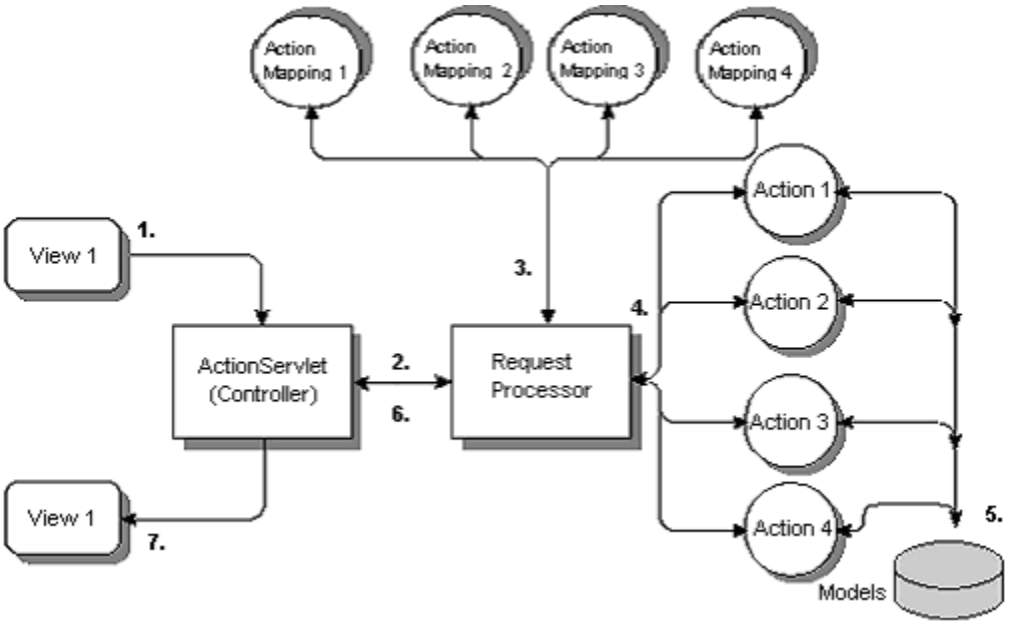
Para cada requerimiento, el *ActionServlet* llama al método process(*HttpServletRequest*, *HttpServletResponse*). Este método simplemente determina que módulo debe atender el requerimiento y delega en su *RequestProcessor* el control.

El *RequestProcessor* es el centro donde se procesan las demandas. El método *process* invoca una secuencia de métodos, delegando de esta manera el procesamiento del requerimiento. Para esta tesis sólo tiene relevancia uno de estos métodos, el *processPreprocess*. Este es uno de los métodos que se pueden re-implementar en las subclases según la especificación de Struts. Este método permite continuar procesando el requerimiento en forma más exhaustiva.

El *RequestProcessor* determina a través de los *ActionMappings* que *Action* debe atender el requerimiento. Esto es posible gracias a que los *ActionMappings* contienen información que identifica que *Action* se corresponde con cada requerimiento. Las propiedades del *ActionMapping* son (entre otras):

- type – nombre de la clase *Action* usada para este mapping.
- path – La url que machee con este mapping.
- forward – La url que toma el control cuando finaliza el mapping.

Una vez identificado el *Action*, el *RequestProcessor* le delega el control para que siga la ejecución del requerimiento. El siguiente dibujo clarifica la secuencia del proceso:



1. El requerimiento es atendido por el *ActionServlet*.
2. El *ActionServlet* delega el control al *RequestProcessor*.
3. El *RequestProcessor* consulta a los *ActionMappings* para determinar el *Action* correspondiente.
4. El *RequestProcessor* pasa el control al *Action* adecuado.
5. Una vez que el *Action* tiene el control realiza las operaciones correspondientes en el modelo y devuelve el control al *RequestProcessor*.
6. El *RequestProcessor* pasa el mando al *ActionServlet*.
7. Se forwardea a la siguiente vista.

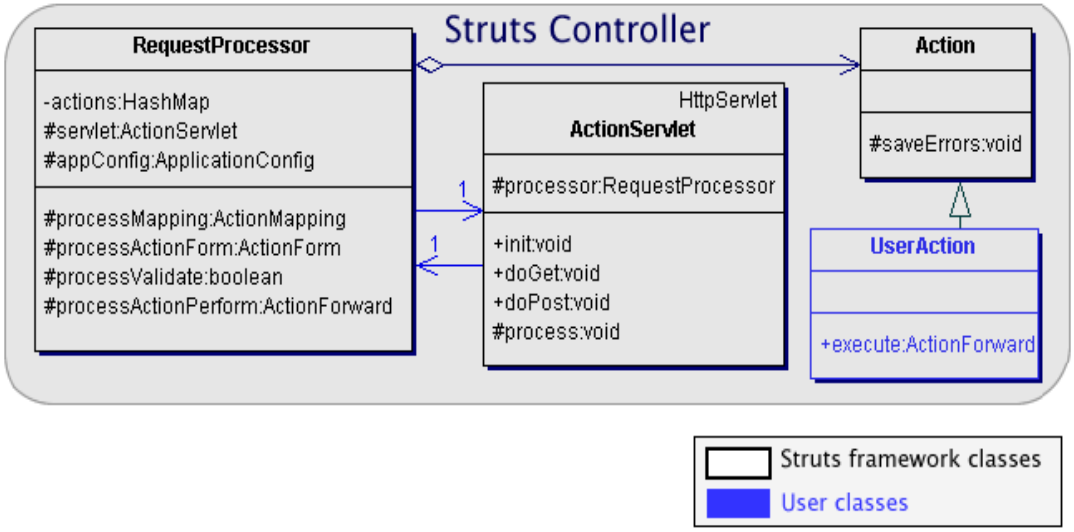
Para terminar la clase *Action* define dos métodos que se ejecutan dependiendo del ambiente del servlet (La mayoría de las aplicaciones sólo usan la versión *HttpServletRequest*.)

- public ActionForward execute(ActionMapping mapping, ActionForm form, ServletRequest request, ServletResponse response) throws Exception;

- public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response) throws Exception;

La función de una clase *Action* es procesar un requerimiento. Para ello se vale del método *execute*, que como resultado del mismo devuelve un objeto *ActionForward*, que será el encargado de identificar quien debe tomar el control.

Para cerrar la explicación del componente controller nos parece adecuado delimitar el alcance del framework, y la instanciación del usuario, con el siguiente ejemplo:



4.1.2 Archivos de Configuración de Struts

Los archivos de configuración de Struts están compuestos de elementos estáticos y dinámicos. Generalmente los componentes estáticos se configuran una sola vez y permanecen sin modificación. Entre estos se encuentran el *controller*, los *message-resources*, los *plug-in* y los *data-sources*. Todos estos están configurados en la distribución del framework de Struts. Entre los elementos dinámicos se encuentran los *action-mapping* y los *form-bean* que definen el comportamiento de la aplicación.

De los elementos estáticos, nos centralizaremos sólo en el controller para lograr una mayor comprensión cuando se explique la extensión del mismo.

En el archivo de configuración web.xml se especifica el servlet de control y sus parámetros de inicialización. Esto se realiza de la siguiente manera:

```
<web-app>
…
    <servlet>
        <servlet-name>action</servlet-name>
        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
        <init-param>
            <param-name>config</param-name>
            <param-value>/WEB-INF/struts-config.xml</param-value>
        </init-param>
    </servlet>
…
</web-app>
```

Como se puede apreciar, es en este archivo donde se especifica el nombre del servlet, que en este caso es **action**, y la clase a la cual pertenece. También se ejemplifica la forma en que se configuran sus parámetros. En este ejemplo se especifica el parámetro llamado **config**, que establece que el archivo de configuración de la aplicación es **/WEB-INF/struts-config.xml**. Existen otros parámetros que se establecen de la misma forma, en especial el mapping, que permite especificar la clase *ActionMapping* correspondiente a la aplicación.

Muchos de los parámetros del controlador se definen a través de los parámetros de inicialización en el archivo web.xml. El elemento <*controller*> por el contrario se define en el archivo struts-config.xml, para permitir diferentes módulos en una aplicación y configurarlos en forma aislada.

La definición por default del elemento <*controller*> en el archivo struts-config.xml es la siguiente:

```
<struts-config>
...
    <controller processorClass="org.apache.struts.action.RequestProcessor" />
...
</struts-config>
```

Si se desea cambiar el controlador, hay que especificar la nueva clase que se encargará de procesar los requerimientos, esta deberá ser subclase de *RequestProcessor*.

Para definir el mapping, en el archivo web.xml, hay dos posibles maneras de especificar las URLs que serán procesadas por el servlet controlador. Una es machear por un prefijo y la otra machear por una extensión.

En el caso de machear por un prefijo, todas las URLs tienen que comenzar con dicho prefijo. Esto se especifica de la siguiente manera:

```
<web-app>
…
    <servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern>/do/*</url-pattern>
    </servlet-mapping>
```

```
…
</web-app>
```

En el caso de tener un requerimiento, por ejemplo /logon la URI correspondiente al mismo, quedaría de la siguiente manera:

```
http://server/myApplication/do/logon
```

En este ejemplo **/server** es la dirección donde se encuentra el Server que contiene la aplicación. La parte de **/myApplication** es el path donde se encuentra la aplicación. Por otro lado **/do** detalla que servlet atenderá el requerimiento. Por último **/logon** es el nombre del requerimiento que se quiere ejecutar.

En el caso de tener un macheo por extensión, la URI finaliza con un conjunto de caracteres definidos. Por ejemplo el procesamiento de las JSP es mapeado al patrón *.jsp. Para usar la extensión *.do se tendría que especificar de la siguiente manera:

```
<web-app>
…
      <servlet-mapping>
            <servlet-name>action</servlet-name>
            <url-pattern>*.do</url-pattern>
       </servlet-mapping>
…
</web-app>
```

La URI para el requerimiento /logon, quedaría de la siguiente manera:

```
http://server/myApplication/logon.do
```

Por otra parte, vale aclarar que los *Action* de la aplicación se configuran en el struts-config.xml, fijando el mapping con la URL y los posibles forwards. Esto se realiza de la siguiente manera:

```
<struts-config>
...
      <action-mappings>
      ...
            <action      path="/logon"
                         type="myApplication.Logon">
                         <forward name="success" path="/index.jsp"/>
             </action>
       ...
       </action-mappings>
...
</struts-config>
```

El elemento **action** determina la URL con el atributo **path**, y con el atributo **type** establece la clase *Action* que macheará con la correspondiente URL. El elemento **forward** especifica a quien se le pasará el control una vez que el *Action* se haya ejecutado.

Un *ActionMapping* toma la información del tag action en tiempo de ejecución. Esto lo realiza mediante reglas de configuración y especificaciones de dtd. Las reglas están especificadas en la clase *ConfigRuleSet*, esta clase realiza el parseo del archivo struts-config.xml. En caso de necesitar otros elementos en el tag action además de los que provee el framework, se pueden crear *ActionMappings* acordes a las necesidades de las aplicaciones. Esto permite poder configurar más información.

Struts tiene configurado 6 librerías de tag especificadas en el archivo de configuración web.xml. Mostramos a continuación un ejemplo de cómo se definen:

```
<taglib>
…
        <taglib-uri>/tags/struts-html</taglib-uri>
        <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
…
</taglib>
```

En este ejemplo la librería **struts-html** contiene tags que se utilizan para crear interfaces basadas en html.

También provee las siguientes librerías de tags:

- struts-bean, permite el acceso a las propiedades de los beans.
- struts-logic, provee operaciones lógicas, como iteraciones, condiciones, etc.
- struts-template, permite definir mecanismos de template.
- struts-tiles, permite combinar varias vistas, en una vista compuesta. Es parecido al struts-template, pero con más características en los tags.
- struts-nested, es una extensión de otras librerías de struts que permite usar objetos anidados.

Algunos de los beneficios que trae aparejado el uso de estas librerías son: reducir y eliminar los scripting tag, permitir portabilidad, separar la lógica del contenido, encapsular comportamiento, etc. Además de usar los tags proporcionados por las librerías de Struts, es sencillo crear tags específicos para la aplicación que se está desarrollando, para facilitar la creación de la interfaz del usuario.

4.2 Extensión de Struts para soportar Ubicación

4.2.1 Ampliación conceptual del MVC para aplicaciones de Hipermedia Física

Nuestro principal objetivo en esta tesis es ampliar la funcionalidad existente para aplicaciones de Hipermedia Tradicional agregando elementos para soportar la parte de ubicación. De esta manera, planteamos reutilizar Struts para resolver los requerimientos de Hipermedia Tradicional e incorporamos nuevos componentes que resuelvan las cuestiones de Hipermedia Física. Esto permite que si uno quiere migrar una aplicación de Hipermedia Tradicional, a una aplicación de Hipermedia Física, solo tenga que agregar la parte de ubicación y reutilizar todo lo que ya tenía desarrollado para la aplicación.

Para esto, comencemos a analizar más profundamente el patrón de diseño MVC en aplicaciones web:

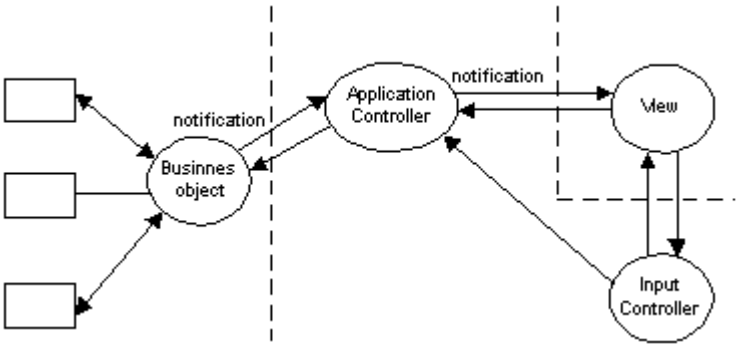


Figura 13: MVC en aplicaciones web [29].

En este esquema, se aprecia la parte de la vista, encargada de la presentación de la información, la parte del controlador que en este caso está compuesta por el InputController y el ApplicationController, encargados de la traducción de la entrada del usuario y del nexo con la aplicación respectivamente y por último la parte del modelo conformada por los objetos de la aplicación que manejan la lógica del negocio.

Si nos referimos a aplicaciones de Hipermedia Física hay que analizar en este punto cuales de las divisiones del MVC tendrían que ser ampliadas para soportar ubicación.

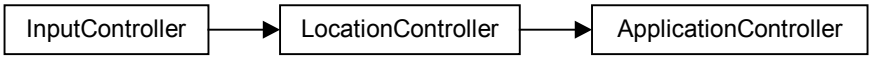
Como ya se mencionó en 3.3. la parte que habría que ampliar sería el controlador. Como se especificó en ese punto se necesita un objeto que se encargue de identificar que tipo de pedido es, es decir, si está asociado a ubicación o no. En el caso que esté relacionado con ubicación tendrá que tener un procesamiento especial. En base a la figura 13 vemos que hasta este punto el controlador está compuesto por dos componentes principales:

- El InputController que determina los mecanismos que se utilizan para el paso de parámetros, extrae cualquier información importante del requerimiento y coopera con el ApplicationController para determinar el *Action* a ejecutar, y lo invoca en el contexto apropiado.

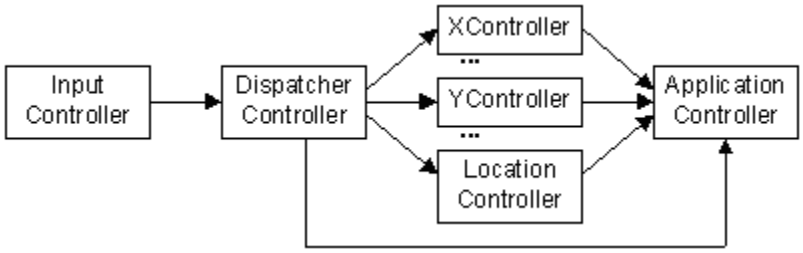
- El ApplicationController que coordina la lógica relacionada al flujo de la aplicación, maneja los errores, mantiene los estados(long-term- states) y determina que vista mostrar.

El comportamiento que se busca agregar estaría más relacionado con el InputController ya que implica de alguna manera resolver un parámetro del requerimiento en este caso, la ubicación o identificación del objeto físico en cuestión. Pero a la vez también, es evidente que este comportamiento es propio de aplicaciones de Hipermedia Física y no es un comportamiento general. Si le agregáramos la responsabilidad de manejar este parámetro al InputController estaríamos forzando a que toda la lógica necesaria para incluir nuevas características a los pedidos sean resueltas

por este componente, lo que es claramente engorroso y poco extensible. Es aquí donde proponemos un nuevo componente denominado LocationController.
Este nuevo componte sería el encargado de resolver aquellos parámetros implícitos o explícitos que correspondan a requerimientos con ubicación. De esta manera el manejo de parámetros tradicional se seguiría resolviendo por el InputController que delegaría en este nuevo componente la carga de resolver las nuevas gestiones pertinentes a ubicación.

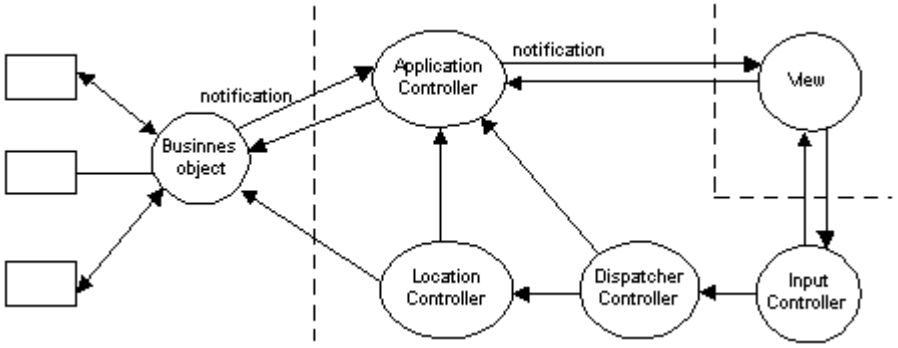


Si bien el LocationController estaría resolviendo el problema, no alcanza con agregar solamente este componente. Pensemos por ejemplo, en un nuevo tipo de aplicación que necesite resolver alguna otra "cuestión de Pre-Procesamiento". Para ello necesitaría entonces otro nuevo controlador y que este se inserte de alguna manera a la secuencia de colaboración recién mencionada. Para resolver esta complicación decidimos agregar un elemento intermedio (DispatcherController) que sea el encargado de distribuir de manera apropiada la secuencia de colaboración entre dichos controladores. Es decir, dependiendo la naturaleza de la aplicación establece que controlador será el colaborador del InputController.



En el caso de tener requerimientos que no necesiten resolver ningún pre-procesamiento, el DispatcherController pasará el control directamente al ApplicationController.

Por lo tanto el diagrama original del MVC en la web se puede ampliar de la siguiente manera para lograr representar el diseño que abarca aplicaciones de Hipermedia Física:



El DispatcherController será el encargado de analizar que tipo de pedido es. En el caso de que sea un pedido solamente digital, pasará el control al ApplicationController. En cambio si es un requerimiento con ubicación el DispatcherController otorga el control al LocationController. El cual será el encargado de analizar las cuestiones pertinentes a ubicación.

Luego de este análisis podemos concluir entonces, que se necesitan dos nuevos componentes para lograr la extensión buscada del MVC. A saber: El DispatcherController y el LocationController. Veamos un poco más a fondo la funcionalidad que tendrá que desarrollar cada uno.

El DispatcherController, como bien lo indica su nombre, será el encargado de distribuir los pedidos o demandas que lleguen. Para esto analizará alguna característica del pedido, la cual debe indicar si se trata de un pedido común o si implica ubicación. Una vez que identifica la naturaleza del pedido, transferirá el control al componente adecuado(ApplicationController / LocationController*)* para que siga la ejecución.

El LocationController, sólo se activa para analizar un pedido o demanda que implica ubicación. Por lo tanto tendrá que obtener esa ubicación ajustándose a las técnicas del sistema de localización que utilice la aplicación. De aquí en adelante continúa el procesamiento como si fuera un pedido común. Para lograr esto el LocationController cede el control al ApplicationController. Cabe aclarar que este último trabaja de manera independiente, sin conocer la naturaleza del pedido.

Estos dos nuevos componentes son el punto clave para la extensión propuesta en esta tesis. Hasta aquí se ha intentado demostrar la justificación conceptual que nos ha impulsado a agregar dichos elementos. De ahora en más nos dedicaremos a analizar como llevar a cabo estas ideas extendiendo el Framework de Struts.

4.2.2 Implementación en Struts del MVC para aplicaciones de Hipermedia Física

Como primera medida hay que analizar las posibles formas de ampliar dicho framework. Según la documentación existente, hay tres maneras posibles de extender Struts:

- Crear un *PlugIn,* esta opción se utiliza si se quiere ejecutar alguna lógica comercial al iniciar o terminar la aplicación.

- Crear un *RequestProcessor*, si se quiere ejecutar alguna lógica comercial durante la fase de procesamiento del requerimiento.

- Se puede extender el *ActionServlet,* si se quiere ejecutar una lógica comercial al iniciar o terminar la aplicación, o durante el procesamiento del requerimiento. Pero sólo debe usarse en los casos dónde ni los *PlugIn*, ni los *RequestProcessor* pueden cumplir la funcionalidad deseada.

Analicemos que es lo que tenemos que extender y cuales de las formas antes mencionadas nos son útiles. Tenemos que lograr tener la funcionalidad detallada para el DispatcherController y el LocationController.

Para el DispatcherController, ninguna de las extensiones detalladas anteriormente resuelven su funcionalidad. Pero investigando como funciona Struts descubrimos que permite tener más de un servlet de control, y brinda soporte para lograr que estos trabajen en forma independiente. Esto lo lleva a cabo a través de URLs disjuntas. Es decir, cada requerimiento se identifica por una URL y según la estructura de esta última se corresponde con un servlet de control. Resumiendo, dependiendo que URL se utiliza, se pasa el flujo de control al servlet correspondiente.

Esta funcionalidad nos permite simular el funcionamiento del DispatcherController. Para lo cual se podría tener una estructura de URL para los pedidos tradicionales y otra para los requerimientos que implican ubicación. De esta manera logramos que Struts nos brinde el soporte para derivar según la URL, el control al servlet correspondiente.

Struts ya tiene un formato para las URL que implican pedidos tradicionales, las cuales se asocian al servlet *ActionServlet*. Como decidimos mantener la funcionalidad de Struts esto permanecerá sin cambios en nuestro framework. Lo que agregaremos es la parte de las URLs que implican ubicación. Estas se asociarán con un nuevo servlet, denominado *LocationActionServlet.*

Lo antes mencionado funciona de la siguiente manera, si es una URL con el formato tradicional, será atendida por el *ActionServlet*. Por el contrario si se trata de una URL con el formato correspondiente a ubicación, el control se parará al *LocationActionServlet*. Esto en el dibujo de la extensión se

corresponde a la parte donde el DispatcherController, delega el control al LocationController o al ApplicationController.

4.2.3 Especificación del nuevo Servlet de Control

La especificación del nuevo servlet de control, se realiza en el archivo de configuración web.xml de la misma manera que está detallado el *ActionServlet* en la distribución de Struts. Quedando dicho archivo de la siguiente manera:

```
<web-app>
...
    <servlet>
        <servlet-name>action</servlet-name>
        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>location</servlet-name>
        <servlet-class>org.apache.struts.location.LocationActionServlet</servlet-class>
    </servlet>
...
</web-app>
```

Como se puede observar **action** es el nombre del servlet que se corresponde con la clase ***ActionServlet***, ya existente en Struts. Y **location** es el nombre de nuestro servlet de control correspondiente a la nueva clase ***LocationActionServlet***.

Para especificar las estructuras de la URL que se corresponden con el servlet de **location** se detalla de la siguiente manera en el mismo archivo web.xml:

```
<web-app>
...
    <servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>location</servlet-name>
        <url-pattern>/location/*</url-pattern>
    </servlet-mapping>
...
</web-app>
```

Según esta especificación se conserva el formato ***.do** para los requerimientos tradicionales, y en el caso de requerimientos con ubicación se corresponderán con el formato **/location/***. Si por ejemplo se tiene el requerimiento /logon que tiene que ser atendido por el servlet **location**, la URL del mismo será:

```
http://server/myApplication/location/logon
```

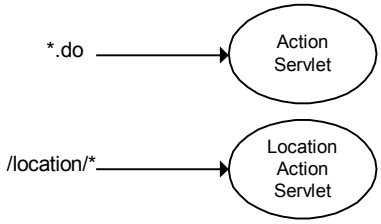
Siendo **/server** la dirección donde se encuentra el Servidor y **/myApplication** el path donde se encuentra la aplicación. Por otro lado **/location** especifica que servlet atenderá el requerimiento, en este caso será el *LocationActionServlet*. Por último **/logon** es el requerimiento que se quiere ejecutar.

En el caso de los pedidos tradicionales, la URL mantendrá la forma provista por Struts. Quedando de la siguiente manera en el caso de tener como requerimiento /logon:

```
http://server/myApplication/logon.do
```

Si bien Struts permite de esta manera manejar más de un servlet de control hay que tener en cuenta que las URLs de cada uno de los servlet tienen que ser disjuntas para que no se generen conflictos.

En conclusión internamente se destinan los requerimientos tanto sea al *ActionServlet* o al *LocationActionServlet* según la especificación del la URL. Gráficamente lo antes mencionado se puede representar de la siguiente manera:

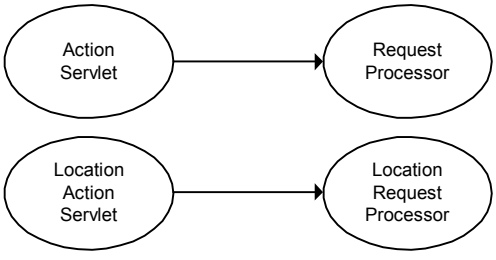


Se logra de esta manera poder tener la funcionalidad del *DispatcherController*, implementada en Struts. Analicemos ahora como llevar a cabo la tarea del LocationController.

4.2.4 Implementación del LocationController

Primero se necesita lograr que los requerimientos que requieran ubicación tengan un tratamiento especial para lograr obtener el objeto físico, esto se ajusta perfectamente a la segunda manera de extensión que provee Struts. Esta alternativa de crear nuestro propio *RequestProcessor*, nos permite darle un tratamiento especial a los requerimientos que impliquen ubicación, lo denominaremos *LocationRequestProcessor.*

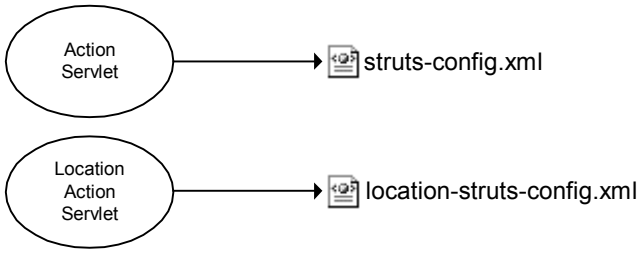
Struts exige que exista una correspondencia uno a uno entre un servlet de control y un *RequestProcessor*. Por lo tanto el *ActionServlet* que existe en Struts se seguirá correspondiendo con el *RequestProcessor,* para garantizar compatibilidad. Y el *LocationActionServlet* se corresponderá con el *LocationRequestProcessor*. Esto se pude representar de la siguiente manera:



De este modo queda bien diferenciada la estructura que se conserva del framework original, de la que incorpora en nuestra extensión.

La idea es lograr que los requerimientos tradicionales sean atendidos por el *ActionServlet*, y este pase el control al *RequestProcessor*. En el caso de que el requerimiento implique ubicación será atendido por el *LocationActionServlet*, y este lo delegará al *LocationRequestProcessor*.

Para especificar la correspondencia de cada servlet de control con su *RequestProcessor*, se debe detallar en el archivo de configuración de cada servlet cual será el *RequestProcessor* asociado. En el caso del *ActionServlet*, Struts provee el archivo de configuración struts-config.xml como se mencionó en 4.1. Para el *LocationActionServlet*, decidimos crear un archivo de configuración, que optamos por denominar location-struts-config.xml. Por lo tanto cada servlet de control tiene su archivo de configuración correspondiente, quedando de la siguiente manera:



La configuración a través de un archivo trae aparejado como beneficio desacoplar la aplicación de su entorno de desarrollo. Es decir, en lugar de codificar las configuraciones mediante clases, se especifican en archivos de configuración. De esta manera, la sustitución de un componente por otro que cumpla la misma interfaz, sólo requerirá cambiar el nombre del elemento en el archivo. De esta manera se evita tener que compilar cada clase cuando se realiza un cambio de configuración, lo que claramente no es viable para el usuario final del framework. Además, permite tener agrupada toda la información configurable, ya que de lo contrario estaría desparramada en las distintas clases. Nuestra decisión de crear un archivo separado del existente, el location-struts-config.xml, se fundamenta en que el desarrollador de la parte física podría ser distinto del de la parte tradicional, y el hecho de compartir un mismo archivo podría ocasionar problemas. Más aún, la configuración de cada *servlet* se establece de manera independiente y esto facilita la depuración o cambios si es que fueran necesarios.

En 4.1 ya se explicó como en el archivo de configuración struts-config.xml se especifica el *RequestProcessor* correspondiente al *ActionServlet*. En el caso de nuestro archivo de configuración location-struts-config.xml la asociación del *LocationActionServlet* y el *LocationRequestProcessor*, queda especificada de la siguiente manera:

```
<struts-config>
...
        <controller  processorClass="org.apache.struts.location.LocationRequestProcessor" />
...
</struts-config>
```

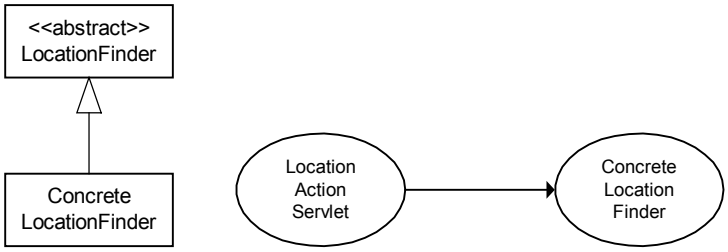
4.2.5 Distribución de responsabilidades para detectar el objeto Físico

Si bien hasta este momento le adjudicábamos al *LocationRequestProcessor* la responsabilidad de detectar al objeto físico, es necesario analizar un poco más en detalle como se resuelve dicha tarea. Es evidente que cada aplicación de Hipermedia Física utilizará el sistema de localización que le resulte más apropiado. Por dicha razón no se puede forzar al *LocationRequestProcessor* la carga de implementar todas las soluciones que esto implica. Queda claro entonces, que definitivamente no será su función resolver concretamente esta tarea. Por el contrario, funcionará de nexo entre el *LocationActionServlet* y un objeto de la aplicación responsable de dicha función. Para permitir la comunicación entre estos dos elementos optamos por definir una Clase llamada *LocationFinder* que es la que establece el protocolo de comunicación con el *LocationRequestProcessor*.

El *LocationFinder* es una clase abstracta, que permitirá al desarrollador instanciar el framework a través de sus subclases. Exige a sus subclases re-implementar dos métodos, uno para lograr identificar el objeto que está frente a una persona (*inFrontOf*) y otro que busca el camino entre dos

objetos físicos (*howToReachFrom*). Esto permite que cada subclase realice las funciones de búsqueda según el sistema de representación de ubicación que se haya elegido.

En tiempo de ejecución el *LocationActionServlet* conocerá a una subclase de *LocationFinder*, es decir, un *LocationFinder* concreto que será el colaborador del *LocationRequestProcessor* para completar su tarea.



El *ConcreteLocationFinder* es configurado en el archivo web.xml, como un parámetro adicional de inicialización del servlet *LocationActionServlet*. Esto se especifica de la siguiente manera:

```
<web-app>
...
   <servlet>
         ...
            <init-param>
                  <param-name>finder</param-name>
                  <param-value>ConcreteLocationFinder</param-value>
            </init-param>
         ...
      </servlet>
…
</web-app>
```

El *ConcreteLocationFinder* tendrá implementado los siguientes métodos según el sistema de localización que se haya adoptado:

- public Object inFrontOfObject(HttpServletRequest obj)
- public Object howToReachFrom(Object from, HttpServletRequest to)

En el primero llega como parámetro el requerimiento, que tiene almacenado la información para buscar el objeto que se encuentra enfrente de la persona. El segundo recibe como primer parámetro el objeto físico que está frente a la persona, y como segundo parámetro tiene el requerimiento que le permite identificar hacia que objeto quiere dirigirse la persona, es decir, que tiene el objeto de origen y destino que permitirán buscar el camino que puede recorrer la persona entre uno y otro.

4.2.6 Almacenamiento de los objetos buscados

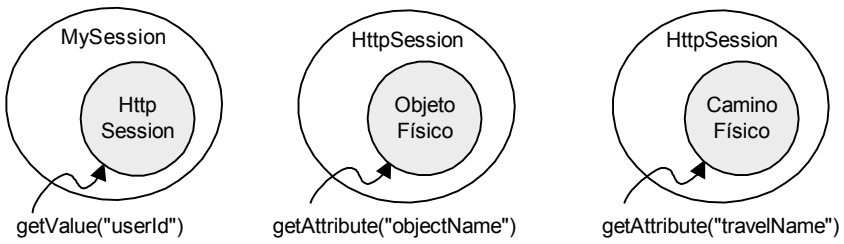
A esta altura ya se distribuyeron las responsabilidades para manejar los requerimientos que implican ubicación. Pero todavía falta analizar otro punto importante que es determinar qué se hace con el objeto físico o la información del camino una vez hallados. Como es de esperar esta información es la que se necesita para terminar con la resolución del requerimiento, y es por ello que decidimos almacenarla en la sesión que mantiene Struts. Para guardar y recuperar objetos de la sesión se necesita asociarles un nombre. Como esta información es sensible a la aplicación resolvimos que el

desarrollador pueda especificar dichos nombres. Para esto agregamos dos parámetros de inicialización del servlet *LocationActionServlet* en el archivo web.xml. Esto permite que luego se puedan recuperar dichos elementos, en el Action y en las JSPs, bajo el nombre que se especificó en el archivo. En el archivo web.xml esto se ve reflejado de la siguiente manera:

```
<web-app>
...
   <servlet>
         ...
            <init-param>
                  <param-name>objectName</param-name>
                  <param-value>Nombre del objeto físico</param-value>
            </init-param>
            <init-param>
                  <param-name>travelName</param-name>
                  <param-value>Nombre del camino a recorrer</param-value>
            </init-param>
         ...
     </servlet>
…
</web-app>
```

El parámetro denominado **objectName**, sirve para especificar bajo que nombre se quiere guardar el objeto físico buscado. Y el parámetro **travelName** permite la especificación del nombre con que se guarda el camino buscado entre dos objetos físicos.

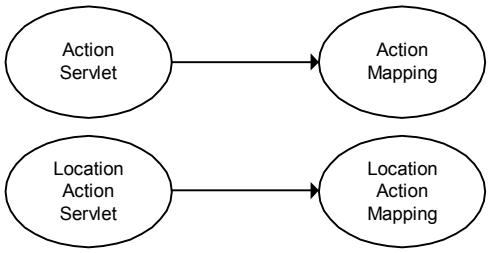
Para entender mejor la manera en que se recuperan los objetos de la sesión veamos el siguiente diagrama:



Primero se necesita recuperar la sesión (en nuestro caso una *HttpSession*) y es a través de esta última se accede a cada uno de los objetos mediante el nombre con que se guardaron.

4.2.7 Análisis de los tipos de pedidos relacionados con ubicación

Todavía nos está faltando en esta secuencia de colaboraciones, determinar de que manera nuestro controlador establece si el pedido es sobre un objeto o pide un camino. Para ello, contamos con los *ActionMappings,* donde se guarda toda la información relacionada con el requerimiento. Struts permite definir *ActionMappings* que tengan información adicional, además de la que provee por default el framework. Por consiguiente, optamos por definir nuestro propio *ActionMapping*, denominado *LocationActionMapping*. El *LocationActionMapping* estará asociado con el *LocationActionServlet*, y el *ActionServlet* seguirá relacionándose con el *ActionMapping* que provee Struts.



El LocationActionMapping, además de tener las propiedades por default del framework, define la propiedad *locationEvent*. La nueva propiedad permite especificar cual de los métodos del buscador deberá invocar el *LocationRequestProcessor*, esto se indica de la siguiente manera:

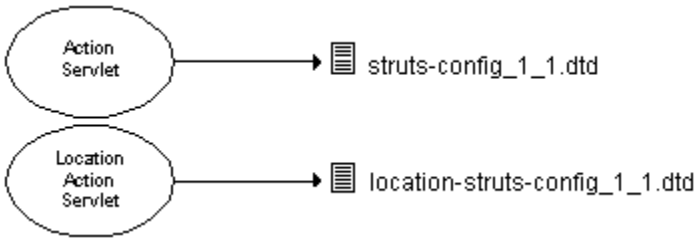
```
locationEvent="inFrontOf"

locationEvent="howToReachFrom"
```

Estas propiedades se definen en el archivo de configuración location-struts-config.xml, siendo este el archivo de configuración del *LocationActionServlet*. Cabe aclarar que esta propiedad está declarada como propiedad obligatoria, ya que sin dicha información, no se puede resolver el pedido. El *LocationActionMapping* lo especificamos como un parámetro de inicialización del *LocationActionServlet*, quedando de la siguiente manera:

```
<web-app>
...
   <servlet>
         ...
            <init-param>
               <param-name>mapping</param-name>
               <param-value>org.apache.struts.location.LocationActionMapping</param-value>
            </init-param>
         ...
      </servlet>
…
</web-app>
```

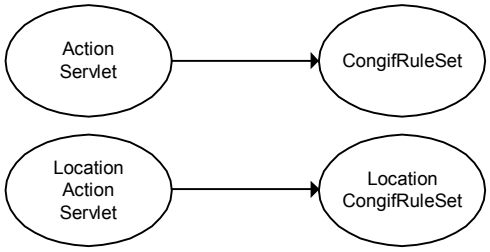
Los archivos de configuración están relacionados con dtds, que determinan la estructura interna que deben respetar dichos archivos. Al agregar una propiedad en el location-struts-config.xml, hay que definir un nuevo dtd, que considere la incorporación de la propiedad en la estructura del archivo. Por lo tanto definimos nuestro dtd, que se asocia con el *LocationActionServlet.*



Al cambiar la estructura del archivo, hay que tener en cuenta que cambian las reglas de parseo del mismo. Por consiguiente, definimos nuevas reglas que permitan el parseo del archivo location-struts-

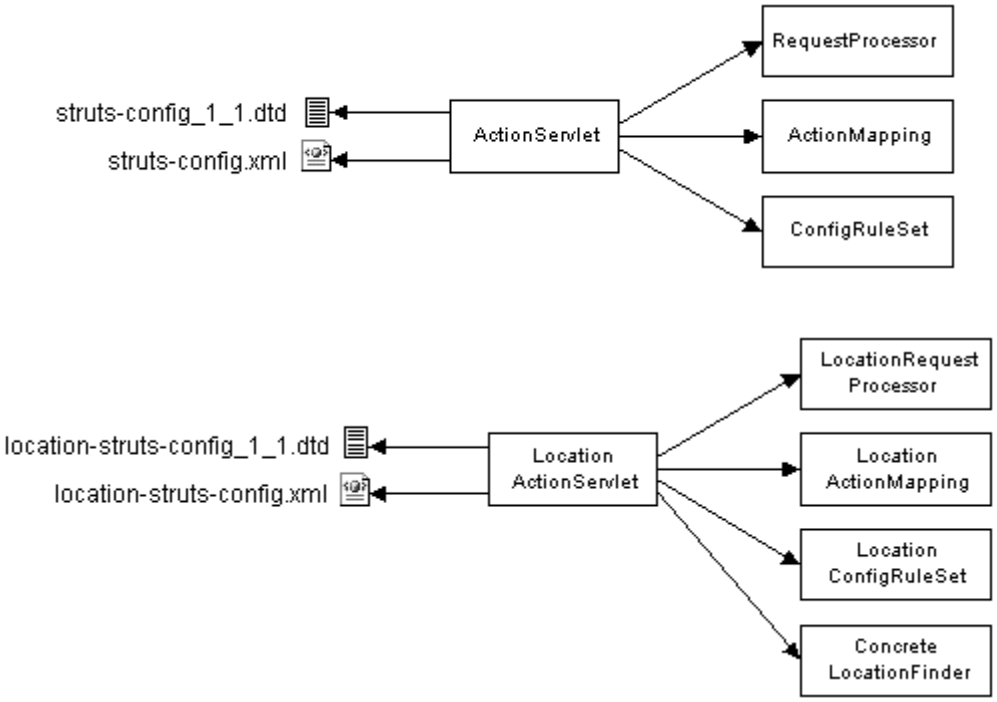
config.xml. En el caso del archivo struts-config.xml se parseará con las reglas provistas por Struts, debido a que no ha sufrido ninguna modificación.

Las reglas de parseo de Struts se encuentran representadas por la clase *ConfigRuleSet*, que está asociada al *ActionServlet*. Para lograr representar nuestro nuevo conjunto de reglas creamos la clase *LocationConfigRuleSet*, que se asocia con el *LocationActionServlet*.

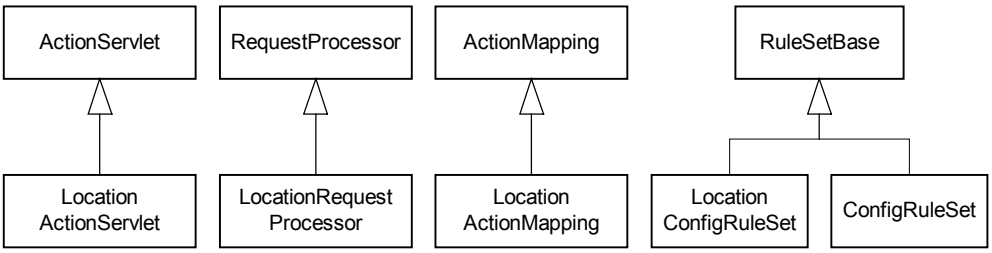


4.2.8 Ampliación de los elementos más destacados

Veamos un diagrama que agrupe todas las relaciones que poseen el *ActionServlet* y el *LocationActionServlet*, y los distintos elementos que fuimos nombrando:



La diferencia fundamental del *LocationActionServlet* es que se relaciona con un *ConcreteLocationFinder*, este es el que permite resolver la parte de ubicación. Como se puede observar, el *ActionServlet* y el *LocationActionServlet* tienen casi las mismas relaciones, salvo que los elementos que se relacionan con el *LocationActionServlet* tienen además un comportamiento específico. Esto ocurre porque respetan los conceptos de los elementos que se relacionan con el *ActionServlet*, y especifican lo necesario para soportar la parte de ubicación. Por consiguiente, son subclases o parte de una misma jerarquía. A continuación presentamos el siguiente diagrama que muestra el esquema de esta porción del diagrama de clases:



Hasta este momento se ha ofrecido una visión superficial de los cambios necesarios. A continuación profundizaremos en los detalles de implementación de los siguientes elementos:

- *LocationActionServlet*
- *LocationRequestProcessor*
- *LocationFinder*
- *LocationActionMapping*
- Archivo de configuración location-struts-config.xml
- Dtd
- *LocationConfigRuleSet*
- *LocationGlobals*
- Modificación en *RequestUtils*

4.2.8.1 LocationActionServlet

El *LocationActionServlet* es subclase de *ActionServlet*, hereda la mayor parte del comportamiento, pero re-implementa y define métodos que se relacionan con la parte de ubicación. La siguiente tabla detalla los métodos que se re-implementaron:

Método	**Explicación**
public void init() throws ServletException	Debido a que el *LocationActionServlet* necesita de nuevos objetos para realizar su función, es claro que su inicialización difiere de la que hereda del *ActionServlet*.
public ActionMapping findMapping(String path)	La asociación con el *LocationActionMapping*, origina el cambio de este método, que devuelve para el *LocationActionServlet*, un objeto *LocationActionMapping*
private void defaultMappingsConfig(ModuleConfig config)	Este método devuelve el default mapping del *LocationActionServlet,* que en nuestro caso necesitamos sea *LocationActionMapping*
protected synchronized RequestProcessor getRequestProcessor(ModuleConfig config) throws ServletException	Este método es el que se utiliza para obtener el *RequestProcessor*, que en este caso será *LocationRequestProcessor*
protected Digester initConfigDigester() throws ServletException	En este método se especifica el nuevo conjunto de reglas de parseo para el archivo de configuración location-struts-config.xml
protected void initOther() throws ServletException	Este método cargará los nuevos parámetros de inicialización. Más concretamente se encargará de inicializar el nombre con que se guardará el objeto físico y el camino, en la sesión y la clase encargada de resolver las cuestiones de ubicación.

Tuvimos que agregar variables específicas para el *LocationActionServlet*, para poder almacenar la información de los parámetros de inicialización. Las variables que surgieron son las siguientes:

Variable	**Explicación**
private LocationFinder locationFinder;	Variable que contiene el *LocationFinder* de la aplicación, es decir, un *ConcreteLocationFinder*.
private String objectName;	Variable que tiene el nombre del objeto físico que se guardará en la sesión
private String travelName;	Variable que tiene el nombre del camino físico que se guardará en la sesión

Para poder especificar el dtd del archivo de configuración surge el siguiente método:

Método	**Explicación**
private void initRegistrations()	Método que permite especificar el nuevo dtd que se utiliza para el archivo location-struts-config.xml

4.2.8.2 LocationRequestProcessor

El *LocationRequestProcessor* es subclase del *RequestProcessor*, y solamente redefine el siguiente método:

Método	**Explicación**
protected boolean processPreprocess(HttpServletRequest request,HttpServletResponse response)	En este método se procesa el requerimiento determinando si se trata de un pedido de búsqueda de un camino o de un objeto. Dependiendo de que tipo de búsqueda se trate, invocará el método adecuado en el buscador. Es acá donde se realiza un pre procesamiento del requerimiento para buscar la parte de ubicación.

4.2.8.3 LocationFinder

Como ya se mencionó anteriormente, el *LocationFinder* es una clase abstracta. El *LocationRequestProcessor* invoca a uno de estos métodos, según lo que esté buscando.

Método	**Explicación**
public Object inFrontOf(HttpServletRequest request)	Método que define el esquema de proceso para los requerimientos sobre objetos físicos.
public Object howToReachFrom(HttpServletRequest request)	Método que define el esquema de proceso para los requerimientos sobre caminos físicos.

Cada uno de estos delega la búsqueda a métodos que tienen que ser re-implentados por las subclases de *LocationFinder*. Una vez que los métodos de las subclases, devuelven el resultado de la búsqueda se almacenan los objetos encontrados, según los nombres que se especificaron en el archivo web.xml.

Los métodos que luego son implementados por las subclases están especificados en el *LocationFinder* de la siguiente manera:

Método	**Explicación**
public abstract Object inFrontOfObject(HttpServletRequest obj);	Método abstracto que se encarga de determinar el objeto físico en cuestión.
public abstract Object howToReachFrom(Object from, HttpServletRequest to);	Método abstracto que se encarga de obtener el camino físico de los objetos en cuestión.

Estos dos métodos son los que realizarán la búsqueda adecuada según el sistema de localización utilizado en las subclases.

Este buscador fue pensado para realizar cualquier tipo de búsqueda según las necesidades de cada una de las aplicaciones que se desarrollen bajo nuestro framework. Las aplicaciones de Hipermedia Física relacionadas con ubicación deben respetar cierto comportamiento de naturaleza propia como son los métodos inFrontOf y el HowToReachFrom. Estos dos métodos son exigidos para cualquier buscador que se desarrolle para nuestro framework. Además se provee, la posibilidad de que cada buscador implemente aquellos métodos de búsqueda que le sean necesarios. De esta manera el buscador se generaliza para realizar cualquier tipo de búsqueda.

Para especificar cualquier otro nuevo método en el buscador que difiera de los exigidos (inFrontOf y el HowToReachFrom) hay que realizar dos pasos.

a. En el archivo de configuración location-struts-config.xml en donde se especifican los location-action, hay que detallar en el atributo locationEvent el nombre del nuevo método. Por ejemplo:

```
<struts-config>
...
    <location-action-mappings>
    ...
        <location-action    path="/NuevoPath"
                            locationEvent="nombreNuevoMetodo"
                            type="miAplicacion.nuevoTipo"/>
     ...
     </location-action-mappings>
...
</struts-config>
```

b. Luego se debe implementar ese método en el *LocationFinder* concreto de la aplicación. Hay que tener en cuenta que cualquier método nuevo que se quiera implementar, tendrá que tener como parámetro el requerimiento. Es decir, la interfaz del método que se tendría que corresponder con el ejemplo antes citado sería:

```
public Object nombreNuevoMetodo(HttpServletRequest req)
```

Estas son las consideraciones para lograr el funcionamiento de cualquier otro método que se necesite para cada aplicación en particular.

De esta manera se logra tener un buscador genérico para cualquier tipo de representación de localización y, además, fácilmente extensible para realizar diferentes búsquedas según sean necesarias.

4.2.8.4 LocationActionMapping

El *LocationActionMapping* es subclase del *ActionMapping*, hereda todo el comportamiento y define la siguiente variable:

Variable	Explicación
protected String locationEvent;	Esta variable permite recuperar el atributo locationEvent del archivo location-struts-config.xml. Los valores que se pueden especificar son: "inFrontOf" o "howToReachFrom".

4.2.8.5 Archivo de configuración location-struts-config.xml

Para evitar confusiones a la hora de configurar los archivos struts-config.xml y location-struts-config.xml decidimos crear tag específicos para este último.

En el archivo location-struts-config.xml los action se configurarán de la siguiente manera:

```
<struts-config>
...
    <location-action-mappings>
    ...
        <location-action    path="/InFrontOf"
                            locationEvent="inFrontOf"
                            type="myApplication.inFrontOf"/>
     ...
     </location-action-mappings>
...
</struts-config>
```

La diferencia con el archivo struts-config.xml es que en este se tienen los tag **action-mappings** y **action** en lugar de **location-action-mappings** y **location-action**. Esto permite la diferenciación a primera vista entre ambos archivos. En el atributo **path** se especifica el nombre que se usará en la URL para invocar a este action. El atributo **locationEvent** permite identificar que tarea debe realizar el *ConcreteLocationFinder*, que en este ejemplo es buscar el objeto que se tiene enfrente. Por último el atributo **type** permite indicar la clase action que atenderá el requerimiento. Para este **location-action** la URL correspondiente al mismo tendrá la siguiente forma:

```
http://server/myApplication/location/InFrontOf?id=1
```

Este ejemplo muestra como se debe invocar un requerimiento que busca el objeto que está frente a una persona. Vale aclarar que **/server** es la dirección donde se encuentra el Servidor, **/myApplication** es el path donde se encuentra la aplicación. Por otro lado **/location** detalla que servlet atenderá el requerimiento, en este caso será el *LocationActionServlet*. El nombre del requerimiento que se quiere ejecutar es **/InFrontOf**. El signo **?** indica que el requerimiento tiene parámetros, en este caso su nombre es **id** y su valor es **1**. Este parámetro permitirá al buscador identificar que objeto debe buscar.

Otro ejemplo interesante es mostrar como se debe configurar esta parte si se quisiera realizar la otra búsqueda, la del camino entre un objeto físico y otro.

```
<struts-config>
...
    <location-action-mappings>
    ...
        <location-action    path="/HowToReachFrom"
                            locationEvent="howToReachFrom"
                            type="myApplication.howToReachFrom "/>
     ...
     </location-action-mappings>
...
</struts-config>
```

Para este caso el **locationEvent** es **howToReachFrom**, y la URL que se corresponde con este **location-action** tendrá la siguiente forma:

```
http://server/myApplication/location/HowToReachFrom?id=2
```

El nombre del requerimiento que se quiere ejecutar es **/HowToReachFrom**. Para dicho requerimiento el parámetro tendrá como nombre **id** y como valor **2**. Este parámetro indica el objeto destino, al cual se quiere llegar desde el objeto que está frente a la persona. El *ConcreteLocationFinder* se encargará de buscar el camino entre el objeto guardado en la sesión y dicho objeto.

4.2.8.6 Dtd

La estructura de los archivos XML se define en un archivo DTD. Debido a que hemos agregado un nuevo archivo XML(location-struts-config.xml), cuya estructura difiere del archivo struts-config.xml original, creamos el DTD asociado.

Los aportes realizados en este archivo (location-struts-config_1_1.dtd)se detallan a continuación:

```
<!ELEMENT location-action-mappings (location-action*)>
<!ATTLIST location-action-mappings      id      ID              #IMPLIED>
<!ATTLIST location-action-mappings      type    %ClassName;     #IMPLIED>

<!ELEMENT location-action (icon?, display-name?, description?, set-property*, exception*, forward*)>
<!ATTLIST location- action      id              id              #IMPLIED>
<!ATTLIST location- action      attribute       %BeanName;      #IMPLIED>
<!ATTLIST location- action      className       %ClassName;     #IMPLIED>
<!ATTLIST location- action      forward         %RequestPath;   #IMPLIED>
<!ATTLIST location- action      include         %RequestPath;   #IMPLIED>
<!ATTLIST location- action      input           %RequestPath;   #IMPLIED>
<!ATTLIST location- action      name            %BeanName;      #IMPLIED>
<!ATTLIST location- action      parameter       CDATA           #IMPLIED>
<!ATTLIST location- action      path            %RequestPath;   #REQUIRED>
<!ATTLIST location- action      prefix          CDATA           #IMPLIED>
<!ATTLIST location- action      roles           CDATA           #IMPLIED>
```

```
<!ATTLIST location- action      scope          %RequestScope;    #IMPLIED>
<!ATTLIST location- action      suffix         CDATA             #IMPLIED>
<!ATTLIST location- action      type           %ClassName;       #IMPLIED>
<!ATTLIST location- action      unknown        %Boolean;         #IMPLIED>
<!ATTLIST location- action      validate       %Boolean;         #IMPLIED>
<!ATTLIST location- action      locationEvent  CDATA             #REQUIRED>
```

En comparación con el dtd denominado struts-config_1_1.dtd, se cambió el elemento **action** que se reemplazó por el elemento **location-action** y el **action-mappings** fue reemplazado por el **location-action-mappings**. El atributo obligatorio **locationEvent** fue la única incorporación, y esta puede apreciarse en la última línea del código anterior.

4.2.8.7 LocationConfigRuleSet

Recordemos que esta clase es la encargada de parsear el archivo location-struts-config.xml cuya estructura la define el dtd explicado en el punto anterior. Como la estructura de este archivo difiere en dos elementos del struts-config.xml y agrega un nuevo atributo no se puede reutilizar la clase *ConfigRuleSet*. *LocationConfigRuleSet* será subclase entonces de *RulSetBase* del mismo modo que lo es *ConfigRuleSet*.

Para implementar el parseo sólo necesitamos cambiar el siguiente código:

```
public void addRuleInstances(Digester digester) {
        …..
        digester.addSetProperties(
                "struts-config/location-action-mappings/location-action");
        digester.addSetNext(
                "struts-config/location-action-mappings/location-action",
                "addActionConfig",
                "org.apache.struts.location.LocationActionMapping");
        digester.addSetProperty(
                "struts-config/location-action-mappings/location-action/set-property",
                "property",
                "value");
        …..
}
```

Esta parte es donde se puede ver como se toman los cambios, ya que se especifican los nuevos tags que tiene el archivo: **struts-config/location-action-mappings/location-action**. Esto quiere decir que el tag **struts-config** tiene un tag anidado **location-action-mappings** y este a su vez contiene otro nivel de andamiento llamado **location-action**. Además, se especifica que el *ActionMapping*, para nuestro trabajo será el ***LocationActionMapping*** que permitirá recuperar el nuevo atributo locationEvent del archivo, y guardará su valor.

4.2.8.8 LocationGlobals

Struts trabaja con una clase denominada *Globals* que posee variables de clase que le permiten tener almacenada información útil para las aplicaciones. Al crear nuevos elementos con nuestro framework se producen conflictos en dos de esas variables, más precisamente en las que almacena información del *RequestProcessor* y del *ActionMapping*. Esto sucede porque el *LocationRequestProcessor* y el *LocationActionMapping*, deben coexistir con el *RequestProcessor* y el *ActionMapping* respectivamente. Esto nos lleva a tener una nueva clase, denominada *LocationGlobals*, que permite

almacenar información relacionada con el *LocationRequestProcessor* y el *LocationActionMapping*, es decir, son variables propias de la parte de ubicación. De esta manera se evitan los conflictos y cuando un requerimiento de ubicación tenga que usar o buscar estos elementos lo hará a través de la clase *LocationGlobals*. A continuación se especifican estas nuevas variables:

Variable	**Explicación**
public static final String MAPPINGS_KEY_LOCATION = "org.apache.struts.action.MAPPINGS_LOCATION";	Esta variable define el nombre con que se guarda en la sesión el *LocationActionMapping*
public static final String REQUEST_PROCESSOR_KEY_LOCATION = "org.apache.struts.action.REQUEST_PROCESSOR_LOCATION";	Esta variable define el nombre con que se guarda en la sesión el *LocationRequestProcessor*

Ambas variables son utilizadas exclusivamente por el *LocationActionServlet*, en los métodos relacionados con el *RequestProcessor* y el *ActionMapping*.

4.2.8.9 Modificación en RequestUtils

Tuvimos que re-implementar un solo método de todo el framework de Struts. El método que re-implementamos se encuentra en la clase *RequestUtils* y es el siguiente:

```
public static String getActionMappingURL(String action, PageContext pageContext)
```

Este método se utiliza para construir las URLs de las JSP. El método original nos agregaba la extensión del servlet que había atendido el pedido a las URLs. Esto nos generaba un conflicto ya que nos modificaba las URLs y estas dejaban de ser coherentes. Por ejemplo si el que había atendido el pedido era el *ActionServlet* a todas las URLs le agregaba el .do, y las URLs relacionadas con ubicación, quedaban de la siguiente manera:

```
http://server/myApplication/location/InFrontOf.do?id=1

http://server/myApplication/location/HowToReachFrom.do?id=2
```

Esto generaba confusión al momento de buscar que servlet debía atender dichos pedidos, porque se tenía el **/location** que corresponde al *LocationActionServlet* y el **.do** que se asocia al *ActionServlet*, los dos al mismo tiempo.

En el caso de que el último servlet que haya atendido el requerimiento fuera el *LocationActionServlet*, entonces se agregaba el /location a todas las URLs, quedando de la siguiente manera:

```
http://server/myApplication/ location/logon.do

http://server/myApplication/location/ location/InFrontOf?id=1

http://server/myApplication/location/ location/HowToReachFrom?id=2
```

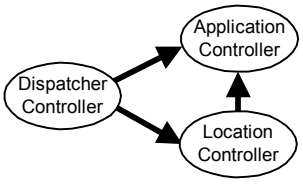
En la primera ocurría el conflicto que se mencionó anteriormente al no saber que servlet debía atender el requerimiento. En el caso de la segunda y tercera son URLs desconocidas al tener **/location/location**.

Estos conflictos nos llevaron a re-implementar este método, eliminando la parte donde se agrega información del servlet que atendió el requerimiento, ya que nos es cierto que el mismo servlet sea el que atiende requerimientos sucesivos.

4.2.9 Relación entre la especificación conceptual y la implementación en Struts

Para comprender mejor todos estos elementos que se estuvieron detallando creemos conveniente detenernos para hacer una pequeña comparación entre los aspectos conceptuales y de implementación.

Recordemos que el componente principal de esta extensión es el controlador. Para ello se propuso realizar la siguiente división conceptual:

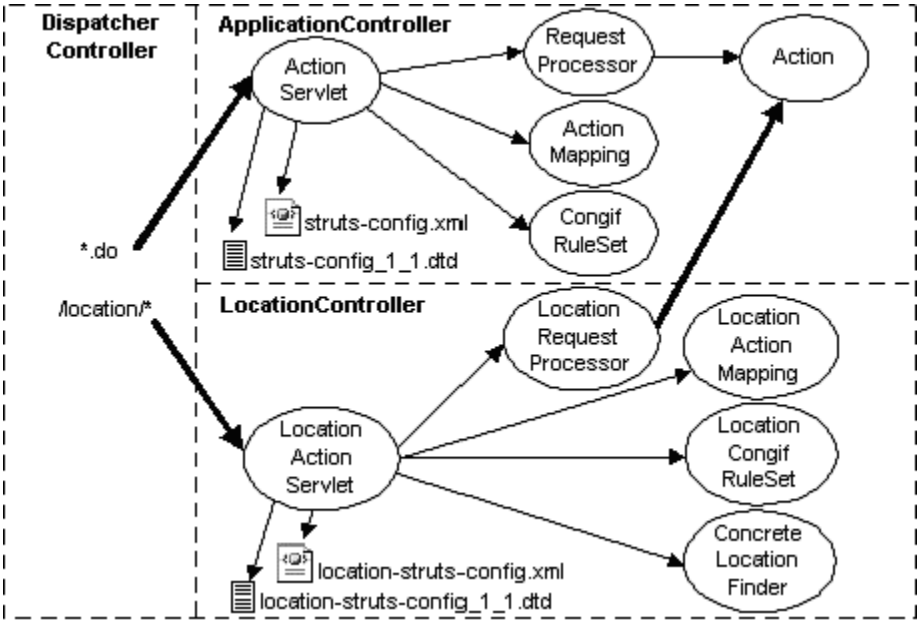


Claramente existen tres componentes bien diferenciados. Como es habitual a la hora de llevar este esquema a una implementación concreta comienzan a surgir nuevos detalles, que a simple vista oscurecen detectar dichos componentes. El siguiente diagrama es el resultado de la implementación de estos componentes en nuestra extensión del framework de Struts.

Dependiendo de la estructura de la URL el DispatcherController transmitirá el flujo de control al ApplicationController o LocationController. El LocationController a través del *Action* pasa el control al ApplicationController.

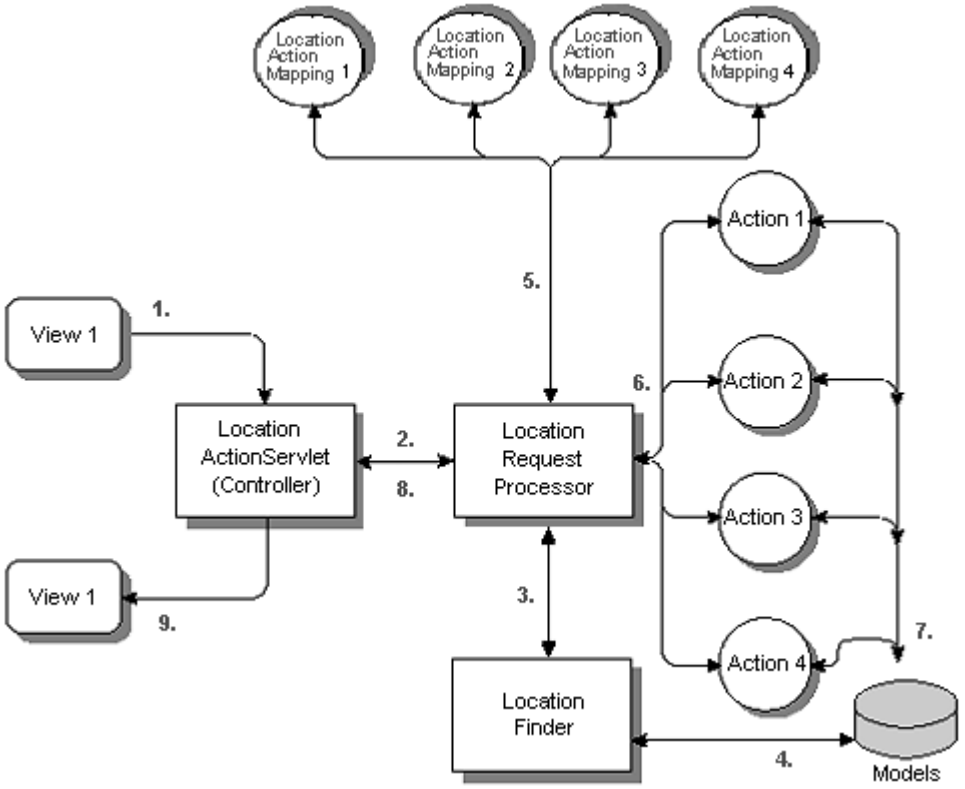
Se puede apreciar que:

- El comportamiento del DispatcherController se resuelve con las estructuras de las URLs
- La funcionalidad del ApplicationController se resuelve con el propio Framework de Struts.
- Las responsabilidades del LocationController se lograron a través de especializaciones de los componentes del Framework de Struts en colaboración con nuevos componentes.



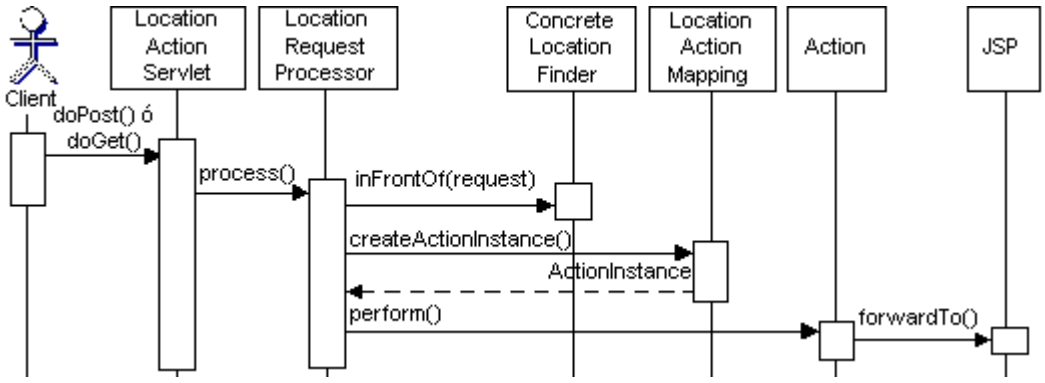
4.2.10 Flujo de control de un requerimiento que implica ubicación

A continuación mostraremos un diagrama que ejemplifica el flujo de control y los elementos intervinientes en la resolución de un requerimiento:



1 El requerimiento es atendido por el *LocationActionServlet*.
2 El *LocationActionServlet* delega el control al *LocationRequestProcessor*.
3 Se pasa el control al *LocationFinder.*
4 El *LocationFinder* localiza la parte física y devuelve el control al *LocationRequestProcessor.*
5 El *LocationRequestProcessor* consulta a los *LocationActionMapping* para determinar el *Action* correspondiente.
6 El *LocationRequestProcessor* pasa el control al *Action* adecuado.
7 Una vez que el *Action* tiene el control realiza las operaciones correspondientes en el modelo y devuelve el control al *LocationRequestProcessor*.
8 El *LocationRequestProcessor* pasa el mando al *LocationActionServlet*.
9 Se forwardea a la siguiente vista.

También se puede ver el flujo de control en un diagrama de secuencia, como se muestra a continuación:



Este diagrama muestra la resolución de un requerimiento del tipo inFrontOf. El ConcreteLocationFinder es específico para cada uno de las aplicaciones según el tipo de sistema de localización que se utilice.

4.2.11 Librería de Tags para aplicaciones de Hipermedia Física

Como se puede apreciar en el siguiente diagrama los custom-tag son el nexo entre la capa de presentación y la de implementación.

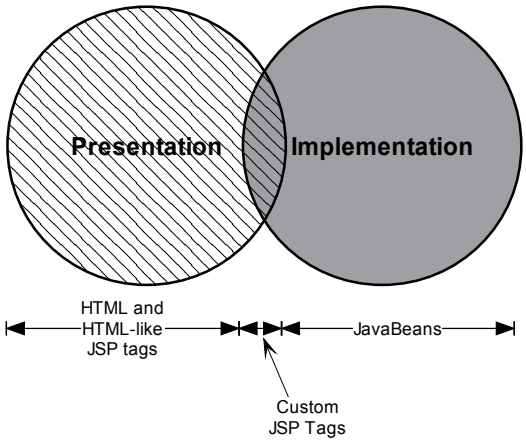


Figura 14: Nexo que realizan los Custom-tag[30].

Como se mencionó en 4.1 Struts provee librerías de tags que traen aparejado muchos beneficios. Esto nos incitó a proveer nuestra propia librería de tags relacionada con la Hipermedia Física.

Estas librerías de tags se incluyen en las JSPs a través de una directiva. A continuación mostramos un ejemplo de como sería la declaración de esa directiva para la librería que explicaremos a continuación.

```
<%@taglib uri="/web-inf/physical-tag.tld" prefix="physical"%>
```

Con esta directiva se establece el prefijo **physical** que se utilizará para referenciar los tag definidos en nuestra librería llamada **physical-tag.tld** ubicada en el path **/web-inf/**.

Con respecto a la información relacionada con Hipermedia Física, podemos destacar los links físicos asociados a un objeto físico, y la información física de ese objeto. Por lo tanto para manejar dicha información surgen dos tags que encapsularán esta información como son: el tag *information* y el tag *links*.

Utilizando estos custom-tags se le permite al desarrollador mantener la funcionalidad existente, además de la posibilidad de trabajar con objetos que no sean sólo beans. Con esto nos referimos a objetos un poco más complejos, pudiendo de esta manera invocar métodos sobre ellos con parámetros y métodos anidados, lo que claramente nos ofrece mayor flexibilidad y funcionalidad. De esta manera la información a mostrar no tiene que ser provista con getters estrictamente. Esto le permite a los desarrolladores realizar por ejemplo filtrados en la información que brinden los objetos físicos.

4.2.11.1 Tag Information

La funcionalidad de este tag será proveer información física de un objeto guardado en una sesión. Para ello la información que se necesita es la siguiente:

- El nombre con el que se guardó el objeto en la sesión.

- Una manera genérica de invocar métodos sobre el objeto guardado.

Para soportar esta funcionalidad optamos por definir el nombre del objeto como un atributo del propio tag (**objectName**) y la especificación de la secuencia de métodos a invocar como tags internos de este (**methodForInformation**). Por ejemplo se podría especificar:

```
<%@taglib uri="/web-inf/physical-tag.tld" prefix="physical"%>
...
<physical:information   objectName="xx">
        <physical:methodForInformation  name="getPosition" />
</physical:information>
```

En este ejemplo puede observarse que a través del atributo **name** del tag **methodForInformation** se especifica el nombre del método que se quiere invocar.

Es evidente que si hablamos de métodos, surge la necesidad de pasar parámetros. Esto se especificará a través de nuevos tags anidados (**parameter**). A continuación mostramos un ejemplo de cómo se podría invocar un método con dos parámetros.

```
<physical:information   objectName="xx">
        <physical:methodForInformation  name="methodName">
                <physical:parameter value="value1" type="type1" />
                <physical:parameter value="value2" type="type2" />
        </physical:methodForInformation>
</physical:information>
```

En este ejemplo observamos que es necesario especificar el valor de cada parámetro mediante el atributo **value** además del tipo del mismo con el atributo **type**. Vale aclarar en este punto que se tendrán tantos tags **parameters** como parámetros tenga el método en cuestión.

Profundicemos un poco más el valor con que debe completarse el atributo **value**. Como ya se explicó en otro punto de la tesis, Struts mantiene una sesión por usuario con objetos pertenecientes a la aplicación. Para recuperar dichos objetos se utiliza un nombre y es precisamente ese nombre el que debe utilizarse como valor del atributo **value**. Notar que el contenido del atributo **type** se corresponderá directamente con el tipo del objeto guardado en la sesión con ese nombre.

Puede ocurrir que la información no sea obtenida de manera directa y se necesite recurrir a otros métodos. Para esto proveemos la posibilidad de tener métodos anidados, esto se detalla de la siguiente manera:

```
<physical:information   objectName="xx">
        <physical:methodForInformation  name="methodName1">
                <physical:parameter value="value1.1" type="type1.1" />
                <physical:parameter value="'value1.2" type="type1.2" />
                <physical:methodForInformation  name="methodName2">
                        <physical:parameter value="value2.1" type="type2.1" />
                        <physical:parameter value="value2.2" type="type2.2" />
                        <physical:methodForInformation  name="methodName3">
                </physical:methodForInformation>
        </physical:methodForInformation>
</physical:information>
```

En este ejemplo se aplica el método **methodName1**, el cual posee los parámetros **value1.1** y **value1.2**, al resultado de eso se le aplica el método **methodName2**, que tiene dos parámetros a saber: **value2.1** y **value2.2**. Por último se aplica el método **methodName3**. La anidación de métodos puede seguir y los mismos pueden tener o no parámetros.

También es importante destacar que se puede listar tanta información del objeto físico como se desee, esto se realiza de la siguiente manera:

```
<physical:information   objectName="xx">
        Info1:
        <physical:methodForInformation  name="methodNameInfo1" />
                <physical:parameter value="value1.1" type="type1.1" />
        </physical:methodForInformation>
        Info2:
        <physical:methodForInformation  name=" methodNameInfo2" />
        Info3:
        <physical:methodForInformation  name="methodNameInfo3" />
                <physical:parameter value="value3.1" type="type3.1" />
                <physical:parameter value="'value3.2" type="type3.2" />
                <physical:methodForInformation  name=" methodNameInfo3.1" />
        </physical:methodForInformation>
        …
</physical:information>
```

El ejemplo anterior lista información relacionada con el objeto físico guardado en la sesión con el nombre **xx**. Notar que se pude agregar información textual para contextualizar, en este caso **Info1:, Info2:** y **Info3:** .

Mediante el uso de estos tres tags se obtiene entonces cualquier información del objeto físico, y se puede realizar cualquier tipo de procesamiento con ella.

4.2.11.2 Tag Links

En el caso del tag links se utilizará para armar los links físicos de un objeto en las JSPs. A continuación detallaremos los elementos que lo componen:

- El tag links posee un atributo llamado **objectName** que especifica el nombre del objeto físico guardado en la sesión, del cual se obtendrán los links.

```
<physical:links   objectName="xx">
```

- El tag anidado **action** permite especificar el action al cual hacen referencia cada uno de los links, en este caso se asume que es un mismo action para todos. El atributo name se completa con el String del path del *Action*, por ejemplo:

```
<physical:action  name="/myApplication/location/HowToReachFrom"/>
```

- Para indicar como obtener a partir del objeto físico la lista de links, se usa el tag **methodForObjectList**. Se completa el atributo **name** con el nombre del método en cuestión:

```
<physical:methodForObjectList  name="relacionesFisicas" />
```

Podría ocurrir que la lista se obtenga mediante filtrados o métodos anidados, o métodos con parámetros. Esto se realiza de la siguiente manera:

```
<physical:methodForObjectList  name="methodName1" />
                    <physical:parameter value="value1.1" type="type1.1" />
                    <physical:methodForObjectList  name="methodName2">
                            <physical:parameter value="value2.1" type="type2.1" />
                    </physical:methodForObjectList >
</physical:methodForObjectList >
```

- A partir de la lista de links obtenida mediante el tag antes mencionado, se especifica como obtener el texto que aparecerá en cada uno de los links. Para esto se utiliza el tag **text** el cual detalla el método o los métodos anidados que serán aplicados a cada uno de los elementos de la lista de links.

```
<physical:text  methodName="descripcion" />
```

En el caso de tener métodos anidados se debe detallar de la siguiente manera:

```
<physical:text  name="methodName1" />
                    <physical:text name="methodName2">
                            <physical:parameter value="value2.1" type="type2.1" />
                    </physical:text >
</physical:text >
```

- Otro requerimiento que se necesita para armar el link físico son el nombre y el valor de los parámetros de dicho action. Para escribir el nombre del parámetro se utiliza el atributo **name** del tag **actionParameter.** Para especificar el valor de dicho parámetro se detalla el método o los métodos anidados que serán aplicados a cada uno de los elementos de la lista de links. Por ejemplo se podría especificar un parámetro llamado id cuyo valor se obtenga de aplicar el método llamado **idNodo** sobre la lista**:**

```
<physical:actionParameter  name="id" >
        <physical:actionParameterValue  methodName="idNodo" />
</physical:actionParameter>
```

También se pueden tener métodos anidados que se especifican de la siguiente manera:

```
<physical:actionParameter  name="id" >
        <physical:actionParameterValue  methodName="methodName1" />
                <physical:parameter value="value1.1" type="type1.1" />
                <physical:actionParameterValue  methodName="methodName2" />
        </physical:actionParameterValue>
</physical:actionParameter>
```

Puede ocurrir que haya más de un parámetro por ejemplo si pensamos en coordenadas x, y que determinen una posición en este caso se deberán especificar dos **actionParameter**, como se muestra a continuación.

```
<physical:actionParameter  name="x" >
```

```
        <physical:actionParameterValue  methodName="methodNameX" />
    </physical:actionParameter>
    <physical:actionParameter  name="y" >
        <physical:actionParameterValue  methodName="methodNameY" />
    </Physical:actionParameter>
```

- El tag **separate** es opcional y permite al desarrollador especificar como quiere que sean separados cada uno de los links que se listarán, por default se tiene un enter (
). Se podría especificar por ejemplo cuatro espacios entre cada uno, esto se detalla de la siguiente manera:

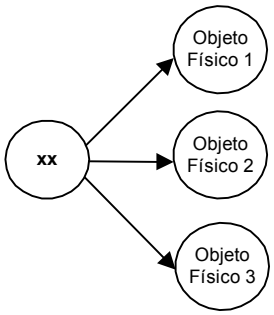
```
<physical:separate  separator="    " />
```

A continuación mostramos un ejemplo completo de cómo especificar todos estos tags en forma conjunta:

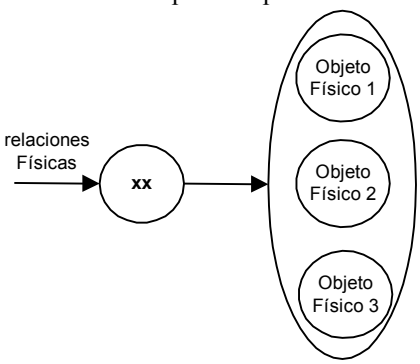
```
<physical:links   objectName="xx" >
        <physical:action  name="/myApplication/location/HowToReachFrom" />
        <physical:methodForObjectList  name="relacionesFisicas" />
        <physical:text  methodName="descripcion" />
        <physical:actionParameter  name="id" >
                <physical:actionParameterValue  methodName=" idNodo" />
        </physical:actionParameter>
</physical:links>
```

4.2.11.3 Ejemplo integrador de los Custom-tag agregados

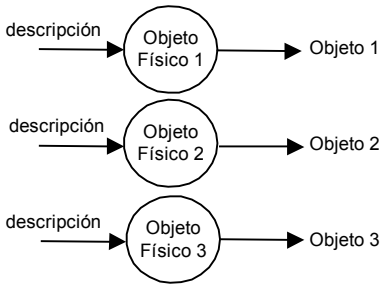
Veamos un ejemplo gráfico de cómo se obtendrían los datos mediante el código antes descripto. Supongamos que desde el objeto **xx** se puede caminar hacia otros tres objetos esto quedaría de la siguiente manera:



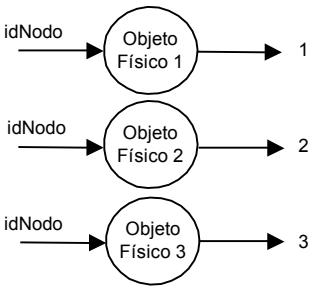
La invocación del método **relacionesFísicas** se puede representar de la siguiente manera:



Como resultado de esta invocación se obtienen todos los objetos físicos relacionados con el objeto **xx**. A Estos objetos obtenidos se les invoca el método **descripción,** quedando lo siguiente:



Como explica el dibujo se obtiene la descripción de cada uno de los objetos. A estos objetos también se les invoca el método **idNodo,** esto puede apreciarse en el siguiente diagrama:



Una vez que se obtuvieron todos estos valores, los mismos son utilizados para describir los links físicos que tiene el objeto físico xx. Esto queda en la JSP de la siguiente manera:

```
<a href==/myApplication/location/HowToReachFrom?id=1> Objeto 1
<br>
<a href==/myApplication/location/HowToReachFrom?id=2> Objeto 2
<br>
<a href==/myApplication/location/HowToReachFrom?id=3> Objeto 3
<br>
```

En la pantalla la persona visualizará los siguientes links:

Objeto 1

Objeto 2

Objeto 3

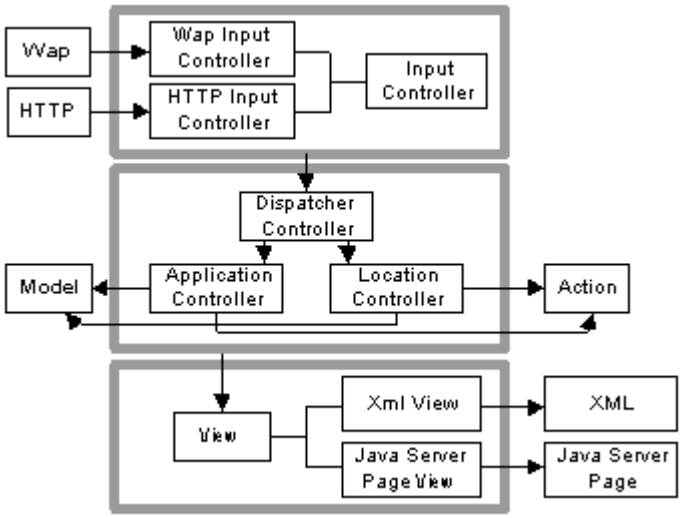
Cada uno de estos al ser clickeados invocan el requerimiento **HowToReachFrom?id=x,** siendo x el identificador del objeto relacionado con el link que se clickeó.

4.2.12 Visualización en los Browsers de las aplicaciones que utilizan nuestro framework

Nuestra extensión es independiente del tipo de Browser que se utilice para la visualización de la aplicación. Es decir, puede utilizarse tanto sea para Browsers Web, como Browsers Wap. Como se mencionó en 4.1 esto es posible gracias a que las JSP son la única parte de framework que posee

código específico que debe interpretar el Browser. Es decir será el desarrollador el encargado de escribir la/s JSP correspondiente/s según a que tipo de Browser esté destinada la aplicación.

En el caso de aplicaciones de Hipermedia Física que se relacionan con ubicación es fundamental que el framework funcione con Browsers Wap, por la naturaleza de este tipo de aplicaciones. Es decir, como se necesitan dispositivos móviles, para que el usuario se traslade con ellos, resulta imprescindible que el framework soporte este tipo de Browsers. En el caso de los Browsers Web, las aplicaciones de Hipermedia Física serían apreciadas como aplicaciones de Hipermedia Tradicional. De más está decir, que sería un poco tedioso movilizar una PC para lograr realizar los recorridos propuestos por las aplicaciones de Hipermedia Física. A continuación mostramos un diagrama que refleja perfectamente como funciona el framework al recibir cualquiera de los inputs:



4.2.13 Pasos para instanciar el Framework

Se detallarán a continuación los pasos que se deben seguir para una buena utilización de nuestro framework.

I. Se debe agregar como librería de cada aplicación nuestro framework que se distribuye en el archivo StrutsLocationFramework.jar.

II. El usuario del framework debe crear su propio buscador, el cual debe ser subclase de *LocationFinder* y debe implementar los siguientes métodos:

Método	**Explicación**
public Object inFrontOfObject(HttpServletRequest req)	En este método se debe realizar la búsqueda del objeto físico que se encuentra frente a la persona. Para lograr encontrar este método llega como parámetro la información necesaria para lograr identificar al objeto físico. El desarrollador de la aplicación conoce que sistema de representación de ubicación utiliza y de acuerdo a este realiza la búsqueda.
public Object howToReachFrom(Object from, HttpServletRequest to)	Lo que tiene que realizar este método es la búsqueda del camino entre dos objetos físicos, el primer parámetro representa al objeto origen, es decir, donde la persona se encuentra ubicada. En el segundo parámetro llega la información necesaria para identificar el destino. Según estos

	dos elementos el desarrollador busca el camino que debe recorrer la persona para llegar al destino.

III. Se debe configurar en el archivo web.xml, la clase del buscador, el nombre con el cual se quiere guardar el objeto físico que está delante de la persona, el nombre con que se guardará el camino entre dos objetos, estos nombres sirven para guardar en la sesión dichos elementos y luego poder ser recuperados por el desarrollador en los *Action* y las JSPs. La parte que se debe configurar es la siguiente:

```
<web-app>
...
   <init-param>
         <param-name>finder</param-name>
         <param-value> clase del buscador especifica de la aplicación </param-value>
   </init-param>
   <init-param>
         <param-name>objectName</param-name>
         <param-value> Nombre con que identificará al objeto físico </param-value>
   </init-param>
   <init-param>
         <param-name>travelName</param-name>
         <param-value>Nombre con que identificará al objeto camino</param-value>
   </init-param>
…
</web-app>
```

Esta especificación permite al framework guardar los objetos una vez que el buscador los encuentra con el nombre que el desarrollador decida. De esta manera no se lo fuerza a usar un nombre impuesto.

IV. En el archivo de configuración struts-config.xml se deben especificar solamente los action que no implican ubicación. La configuración de los mismos se realiza de la misma manera que en una aplicación de Struts común.

V. En el archivo de configuración location-struts-config.xml, se especifican aquellos location-action que involucran ubicación. Además de los atributos que posee Struts, es obligatorio especificar el atributo locationEvent, el cual permite al framework poder invocar el método adecuado en el buscador.

En el caso de buscar un objeto frente a una persona se detallará lo siguiente:

```
locationEvent="inFrontOf"
```

Por el contrario, si se trata de buscar el camino entre dos objetos, se debe especificar:

```
locationEvent="howToReachFrom"
```

VI. Para finalizar se deben definir los Action y las JSP(u otra presentación) específicas para cada aplicación.

Realizando estos 6 pasos se asegura una correcta instanciación del framework.

4.3 Implementaciones de Hipermedia Física

4.3.1 Descripción de un modelo

Para lograr comprender mejor nuestro modelo de una aplicación de Hipermedia Física empecemos por ubicarlo en la arquitectura MVC. Como ya se mencionó en 3.1, los componentes que integran el MVC son tres: el modelo, la vista y el controlador. El siguiente diagrama servirá como base para definir el modelo en cuestión.

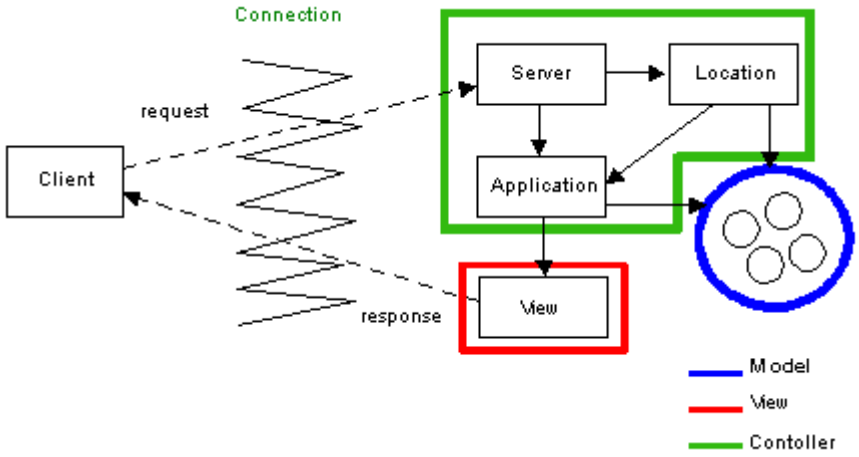


Diagrama 1

El Controller va a estar representado por la instanciación de nuestro framework, el cual se especificó detalladamente en 4.2. La vista estará definida a través de JSPs representativas de la aplicación que se esté desarrollando. Y en el caso del modelo se definirán los objetos de la aplicación que manejarán la lógica del negocio. A continuación, intentaremos ofrecer una visión aproximada del diseño esperado para la especificación de la vista y el modelo.

4.3.1.1 El Modelo

En el modelo se definirán objetos digitales y objetos físicos, incluyendo las relaciones que se le atribuyan a cada uno en particular. Para lograr esto definiremos una estructura genérica que contemple las características de ambos tipos de objetos.

Los objetos digitales serán representados con una abstracción de sus características principales, a saber: nombre, descripción y las relaciones que tienen con otros objetos digitales. Esto se puede apreciar en el siguiente diagrama:

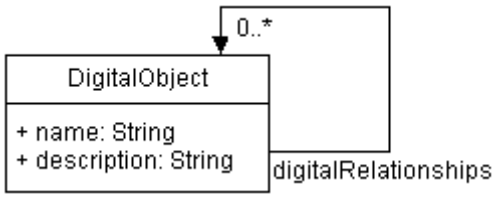


Diagrama 2

Se puede observar que la variable **digitalRelationships** representa las relaciones que tiene el objeto digital con otros objetos digitales. Estas relaciones son las que permiten la navegación digital. Es decir, si se está visualizando información de un objeto digital, se podrá ver solo información de los objetos digitales con los que se relaciona el objeto en cuestión.

En el caso de los objetos físicos tendrán como característica genérica, un **identifier** unívoco según el sistema de posicionamiento elegido para la aplicación, una descripción de su ubicación relativa al mundo real y las relaciones con otros objetos físicos. Esto se puede representar de la siguiente manera:

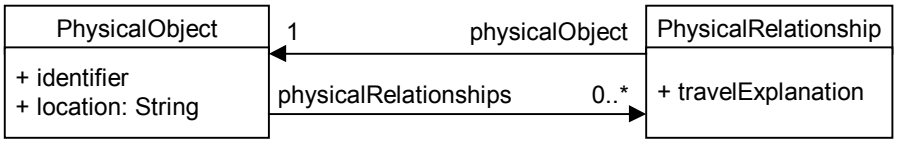


Diagrama 3

Como se puede observar las relaciones físicas están representadas a través de la clase **PhysicalRelationship,** esto ocurre porque la relación debe contar con la especificación del recorrido que debe realizar la persona para llegar del objeto origen al objeto destino. Por lo tanto no se puede representar de la misma manera que se especifican las relaciones digitales, ya que estas no necesitan este nivel de detalle.

En cuanto a la variable **travelExplanation** esta puede tomar diferentes valores dependiendo de cómo se quiere mostrar dicha explicación. Puede elegirse dar una descripción escrita, mostrar el recorrido en un mapa, indicar el camino a seguir mediante un audiovisual, o podría ser una combinación de dichas técnicas. La elección dependerá del tipo de aplicación a realizar y las capacidades de los dispositivos móviles. Al tener diferentes formas de expresar la explicación del camino a recorrer, la variable **travelExplanation** puede modelarse de la siguiente manera:

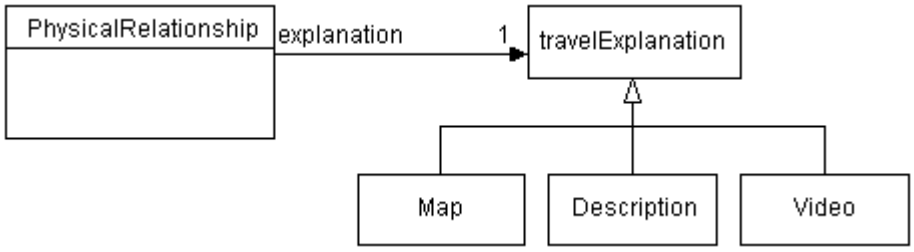


Diagrama 4

Otra variable que puede tomar diferentes valores es el identificador del **PhysicalObject,** ya que dependerá del sistema de localización elegido para el desarrollo de la aplicación. Se podría tener un identificador numérico, o contar con un sistema de GPS, o alguna otra representación. Lo importante que se rescata de esta variable es que debe asegurar la identificación unívoca de los objetos físicos. Una posible modelización de dicha variable puede observarse en el siguiente diagrama:

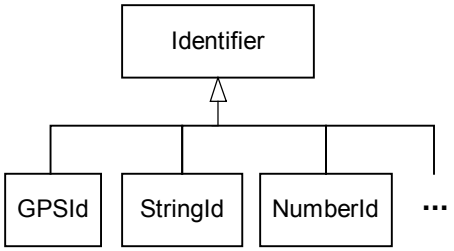


Diagrama 5

Las relaciones de los objetos representan la forma de llegar de un objeto a otro del mismo tipo. Esto permitiría ver a las relaciones como partes de una misma jerarquía, donde cada una de las relaciones se comporta según sus características. Esto se puede diseñar de la siguiente manera:

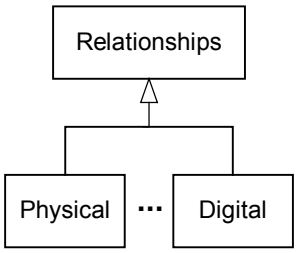


Diagrama 6

Los objetos físicos se asocian generalmente con objetos digitales. Es decir, un objeto de la realidad tiene su representación digital, donde se expresan las características propias del objeto. Esto lo podemos representar de la siguiente manera:

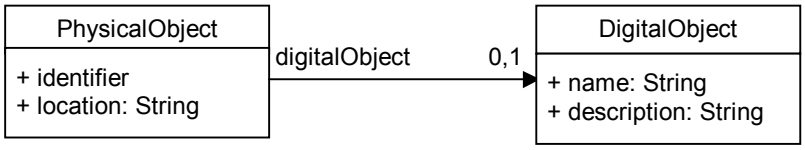


Diagrama 7

La cardinalidad de la relación indica **0,1** ya que se puede tener un objeto digital asociado o no. Puede ocurrir que existan objetos físicos que deben ser representados en la aplicación, pero que la misma no requiera una representación digital de dicho objeto.

A partir de las descripciones anteriores, se pueden unir los componentes genéricos del modelo para conformar el siguiente diagrama:

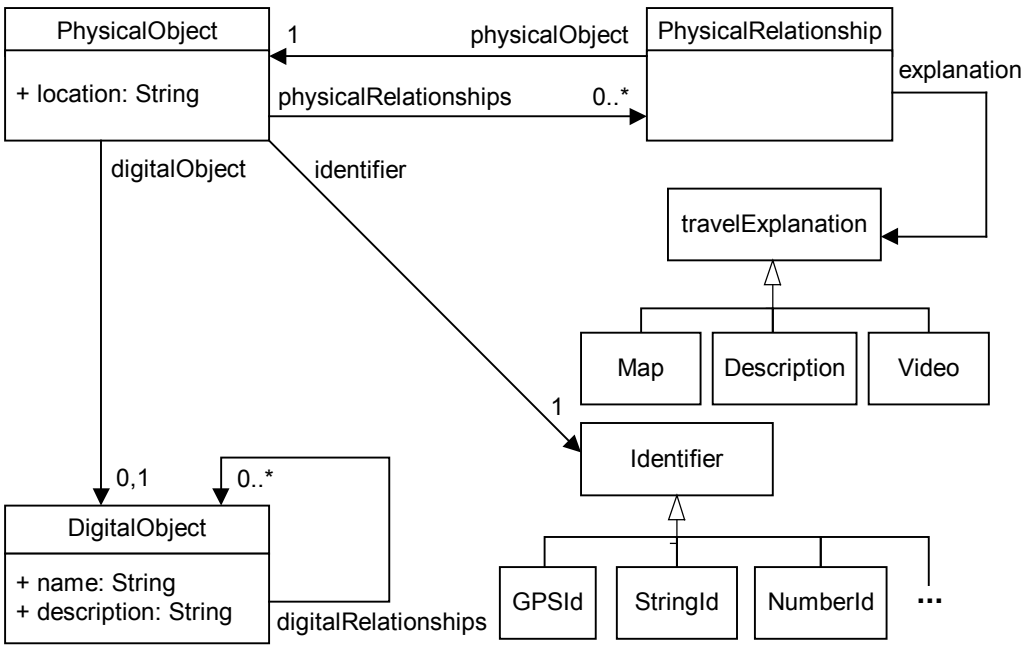


Diagrama 8

En el caso de que se necesite especificar más información de los objetos físicos o digitales se pueden crear subclases de los mismos, logrando tener un modelo más especifico dependiendo de las aplicaciones que se desarrollen.

4.3.1.2 La Vista

Elegimos representar las vistas mediante JSPs, pero se podría haber elegido otro tipo de represtación visual. Cada una de las JSPs contendrá información física y digital. La información se corresponderá en el modelo con los objetos físicos y digitales. Esto se puede observar en el siguiente diagrama:

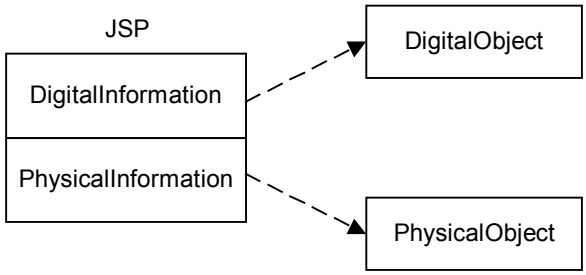


Diagrama 9

Tanto la información física como la digital de las JSPs son completamente independientes entre si. Esto ocurre porque dicha información es obtenida a través de diferentes objetos. Puede ocurrir que la información digital esté relacionada directamente con la información física o no. La información no está relacionada cuando la persona se encuentra en un determinado lugar y está navegando digitalmente a través de las opciones propuestas. Es decir, ve la información física relativa al lugar donde se encuentra y navega digitalmente como si fuera una aplicación de Hipermedia Tradicional.

Las JSPs pueden distribuir ambos tipos de información de diferentes maneras. Esto puede verse en el siguiente diagrama:

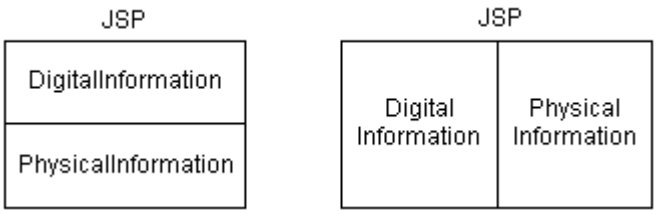


Diagrama 10

Depende del desarrollador la diagramación de las JSPs, quien debe elegir la representación de la información más adecuada al tipo de aplicación que esté desarrollando.

En el siguiente punto se utilizarán los diagramas antes descriptos como base para explicar los patrones de diseño que pueden ser utilizados en las aplicaciones de Hipermedia Física.

4.3.2 Implementación del modelo utilizando patrones de diseño

4.3.2.1 Concepto de patrón de diseño

El principal objetivo de los patrones de diseño es capturar buenas prácticas de diseño que permitan mejorar la calidad del desarrollo de los sistemas, determinando objetos que soporten roles útiles en un contexto específico, encapsulando complejidad, permitiendo flexibilidad y extensibilidad.

Un patrón describe la solución para problemas recurrentes de tal modo que se pueda reutilizar esta solución en contextos diversos o en distintas aplicaciones. Un patrón de diseño es una abstracción de una solución en un nivel alto. Hay patrones que abarcan las distintas etapas del desarrollo; desde el análisis hasta el diseño y desde la arquitectura hasta la implementación.

Los diseñadores de software extendieron la idea de patrones de diseño al proceso de desarrollo de software. Debido a las características que proporcionaron los lenguajes orientados a objetos (como herencia, abstracción y encapsulamiento) permitieron relacionar entidades de los lenguajes de programación a entidades del mundo real fácilmente, los diseñadores empezaron a aplicar esas características para crear soluciones comunes y reutilizables para problemas frecuentes que exhibían patrones similares.

En el ámbito de la orientación a objetos, los patrones se pueden ver como micro-arquitecturas que están más allá de las soluciones sencillas. En este contexto, los patrones se preocupan en especificar cuáles son los componentes de la solución, cuáles son sus responsabilidades, y cuáles son las formas de colaboración entre ellos. De este modo, el conocimiento de los patrones permite que un diseñador inexperto conozca la solución utilizada por un experto para cada par "problema-solución". Por lo tanto los patrones permiten:

- Reutilizar las arquitecturas de software, lo que es mucho más valioso que reutilizar algoritmos o código.
- Mejorar la comunicación del equipo de desarrollo, porque proveen un vocabulario más conciso y un nivel de abstracción más alto.
- Capturar explícitamente el conocimiento implícito utilizado, y no documentado, por los diseñadores expertos.

Citamos algunas definiciones de la literatura sobre patrones de diseño:

- "Los patrones de diseño constituyen un conjunto de reglas que describen como realizar ciertas tareas en el dominio del desarrollo de software". [42]

- "Los patrones de diseño se concentran más en el rehúso del diseño de arquitecturas recurrentes, mientras que los frameworks se centran los detalles de diseño e implementación.". [10]

- " Un patrón describe un problema de diseño recurrente que se origina en situaciones de diseño específicas y presenta una solución para este. ". [6]

Los beneficios que produce el uso de patrones pueden ser medidos en varios sentidos:

- Contribuyen a reutilizar diseño, identificando aspectos claves de la estructura de un diseño que pueden ser aplicados en una gran cantidad de situaciones. La importancia de la reutilización del diseño no es despreciable, ya que ésta nos provee de numerosas ventajas: reduce los esfuerzos de desarrollo y mantenimiento, mejora la seguridad, eficiencia y consistencia de nuestros diseños, y nos proporciona un considerable ahorro en la inversión.
- Mejoran (aumentan, elevan) la flexibilidad, modularidad y extensibilidad, factores internos e íntimamente relacionados con la calidad percibida por el usuario.
- Incrementan nuestro vocabulario de diseño, ayudándonos a diseñar desde un mayor nivel de abstracción.

En 1994, apareció el libro "Design Patterns: Elements of Reusable Object Oriented Sofware"[19] escrito por los ahora famosos Gang of Four (GoF) formada por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. Ellos recopilaron y documentaron 23 patrones de diseño aplicados usualmente por expertos diseñadores de software orientado a objetos. Ellos no son los inventores ni los únicos involucrados en este tema, pero a partir de la publicación de ese libro se empezó a difundir con más fuerza la idea de patrones de diseño. Es en este libro donde los patrones se dividieron en

diferentes categorías de acuerdo a su uso, a saber: patrones de creación, estructurales y de comportamiento.

4.3.2.2 Patrones de diseño J2EE

Existe un conjunto de patrones que pueden ser usados en el contexto del diseño de aplicaciones Java 2 Enterprise Edition, J2EE. Los patrones de diseño J2EE consisten en soluciones recurrentes y documentadas de problemas comunes de diseño de aplicaciones J2EE. Muchas de estas soluciones se basan en los patrones de GoF[19], así que se podrán encontrar soluciones J2EE que son aplicadas al diseño de aplicaciones de software en general.

Los patrones J2EE describen típicos problemas encontrados por desarrolladores de aplicaciones empresariales y proveen soluciones para estos problemas. En esencia, estos patrones contienen las mejores soluciones para ayudar a los desarrolladores a diseñar y construir aplicaciones para la plataforma J2EE.

Con la aparición del J2EE, todo un nuevo catálogo de patrones de diseño apareció. Desde que J2EE es una arquitectura por si misma que involucra otras arquitecturas, incluyendo servlets, JavaServer Pages, Enterprise JavaBeans, y más, merece su propio conjunto de patrones específicos para diferentes aplicaciones empresariales. Los patrones J2EE están divididos en tres capas: presentación, negocios e integración, según el libro "J2EE PATTERNS Best Practices and Design Strategies" [11].

La implementación de patrones J2EE no es trivial en los desarrollos de aplicaciones empresariales. Los desarrolladores necesitan conocer propuestas metodológicas para una correcta implementación de los patrones. Debe quedar claro que los patrones no son las respuestas únicas a un problema, sino que más bien representan una solución estructural y de comportamiento para un problema de diseño recurrente.

4.3.2.3 Patrones de diseño en aplicaciones de Hipermedia Física

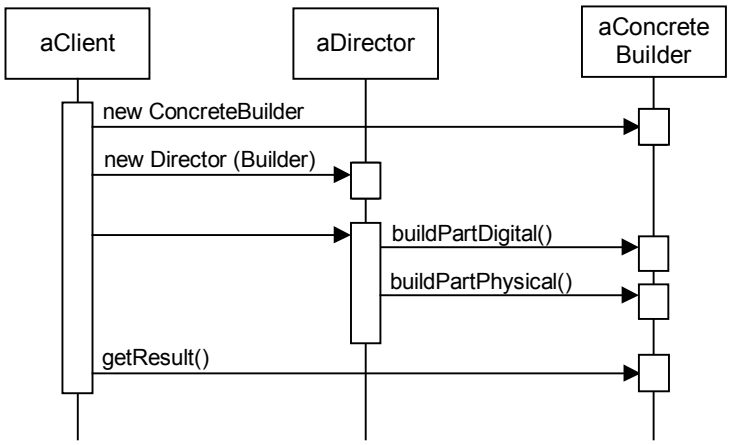
A continuación ofreceremos una lista de los patrones que se podrían utilizar para instanciar nuestro framework bajo buenas práctica de diseño:

4.3.2.3.1 Builder

Intención:
El patrón de diseño Builder[19] separa la construcción de un objeto complejo de su representación, para que el proceso de la construcción permita crear diferentes representaciones. Esto permite: la variación de la representación interna del producto, separar el código de la construcción y la representación, y da un control refinado sobre el proceso de construcción.

Aplicación en Hipermedia Física:
En las aplicaciones de Hipermedia Física se podría usar este patrón para construir las JSPs. Esto es posible gracias a que existe una clara separación en los elementos que integraran las JSPs. Es decir, cada JSP tendrá una parte con información física y otra con información digital. Los dos tipos de información son obtenidos de manera independiente entre si, esto se puede observar en el diagrama 9.

El patrón permitirá construir por un lado la parte digital y por otro la parte física, obteniendo luego el producto final. Es decir, el proceso constará de dos pasos (bien diferenciados) cuyo resultado será la JSP a mostrar. En nuestro ejemplo sólo tenemos dos partes a construir pero dependiendo del tipo de aplicación podrían existir otras divisiones u otro tipo de información. Basándonos en el diagrama de secuencia que describe al patrón, será más fácil comprender su funcionamiento:

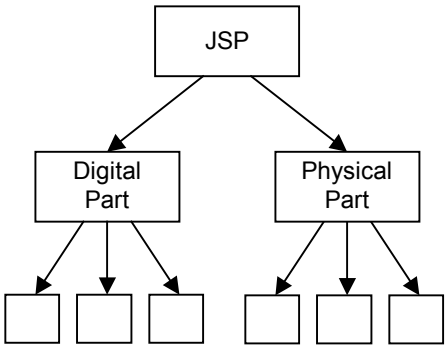


Como resultado se obtiene la JSP, que se formó con la construcción de la parte digital y la parte física.

4.3.2.3.2 Composite

Intención:
Compone los objetos en estructuras de árbol para representar jerarquías enteras o parte de ellas. En el árbol, algunos de los objetos pueden ser los nodos y otros objetos hojas. El patrón de diseño Composite[19] permite a los clientes tratar a los objetos simples y los objetos compuestos de manera uniforme.

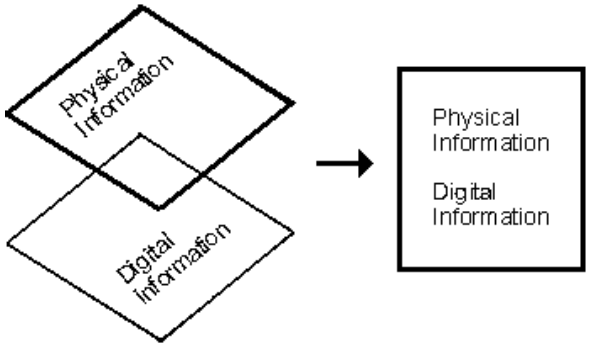
Aplicación en Hipermedia Física:
Si se observa el diagrama 9, se puede apreciar que la JSP esta compuesta de dos sectores bien diferenciados, como son la parte física y la parte digital. Esto permite estructurar la JSP con dos componentes y luego a su vez cada uno contiene elementos internos. Es aquí donde se puede aplicar el patrón Composite[19], quedando la siguiente estructura jerárquica como se muestra a continuación:



4.3.2.3.3 Decorator

Intención:
Une las responsabilidades adicionales de un objeto dinámicamente. El patrón de diseño Decorator[19] provee una alternativa flexible para extender funcionalidad sin mecanismos de herencia. Un objeto es contenido por otro, el cual le agrega nueva funcionalidad.

Aplicación en Hipermedia Física:
Los objetos físicos que tienen una representación digital asociada pueden verse como decoradores de los objetos digitales, esto se muestra en el diagrama 7. A los objetos digitales se les agrega la información física que corresponde.

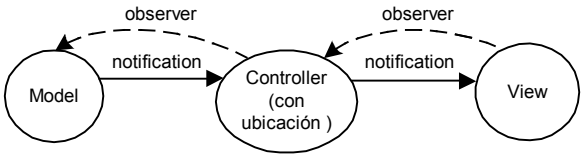


Esto es aplicable a todos los objetos físicos que tienen una asociación digital.

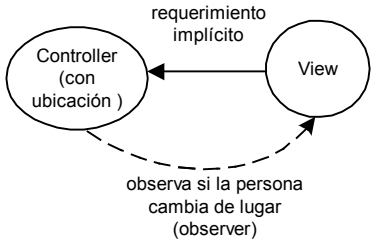
4.3.2.3.4 Observer

Intención:
Define una dependencia entre objetos (de uno a muchos), ya que cuando cambia el estado de un objeto todos sus dependientes son notificados y actualizados automáticamente. El patrón de diseño Observer[19] permite mantener la consistencia entre objetos relacionados, de manera desacoplada entre ellos.

Aplicación en Hipermedia Física:
En la Hipermedia Física se mantiene el concepto de que la vista es observador del modelo, pero se incorpora una nueva observación. Esta consta en que el controlador observa a la vista. Esto ocurre porque la vista depende de la posición del usuario para mostrar la información correspondiente al objeto físico que está frente a la persona. Es decir, cuando el usuario cambia de posición y pasa frente a un objeto físico el controlador es avisado de dicho cambio. Un diagrama que representa las observaciones que existen podría ser:



El diagrama anterior muestra las observaciones que existían en las aplicaciones de Hipermedia. Se puede ver en la siguiente figura la nueva observación que surge en las aplicaciones de Hipermedia Física:

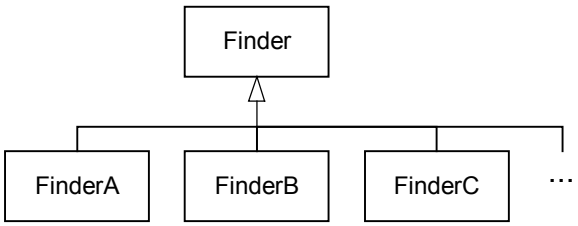


El requerimiento implícito es aquel que se invoca cuando la persona con el dispositivo móvil pasa frente a un objeto físico. A través de algún mecanismo se detecta dicho objeto y se manda un requerimiento implícito sin que la persona lo detecte. Como respuesta a dicho requerimiento la persona visualizará información del objeto que tiene enfrente.

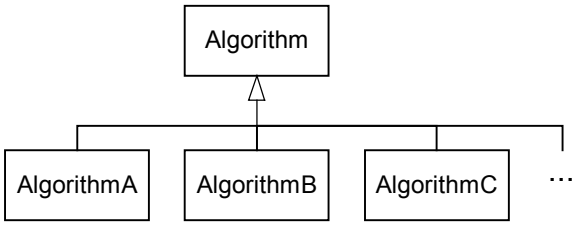
4.3.2.3.5 Strategy

Intención:
Define una familia de algoritmos, encapsulado cada uno, y haciéndolos intercambiables. El patrón de diseño Strategy[19] permite que el algoritmo varíe independientemente de los clientes que lo usan. Permite cambiar fácilmente entre los algoritmos sin ningún condicionamiento. Los algoritmos llegan a una solución para el mismo problema.

Aplicación en Hipermedia Física:
Se puede aplicar este patrón en el buscador que instancia nuestro framework, una opción sería tener una jerarquía de buscadores que sean equivalentes entre ellos. Es decir, cualquiera que se elija en tiempo de ejecución cumplirá la tarea de buscar la información física. El siguiente diagrama ejemplifica lo antes mencionado:



Esto modelaría una estrategia de buscadores. También podría tenerse dentro de un buscador distintos métodos que realicen la búsqueda. Es decir, estrategia de algoritmos de búsqueda. Esto se puede apreciar en el siguiente diagrama:



Otra aplicación de este patrón podría utilizarse al momento de diseñar el sistema de localización. Es decir podría haber diferentes sistemas de localización en la aplicación. Esto se encuentra diseñado en el diagrama 5.

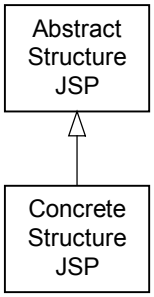
4.3.2.3.6 Template Method

Intención:
Define un esqueleto para un algoritmo, mientras las subclases difieren en alguno de sus pasos. El patrón de diseño Template Method[19] permite a las subclases redefinir ciertos pasos del algoritmo sin cambiar la estructura del mismo.

Aplicación en Hipermedia Física:
Podría usarse este patrón en la construcción de las JSP. Struts provee template de construcción para las JSP. Esto permite aislar la estructura de las JSP, de la información a mostrar. Usando los templates de Struts, se puede cambiar la estructura sin tocar la información a mostrar.

Hay distintas formas de visualizar la información de una JSP. Esto se puede observar en el diagrama 10. Pero podrían existir tantas representaciones como se quisiese. El uso del templates permite cambiar la estructura de forma independiente de la información.

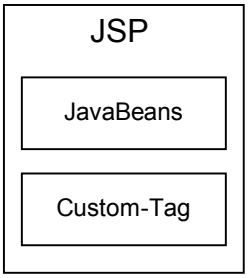
En este caso se estaría usando la estructura existente de Struts, y se instanciaría según la estructura necesaria de cada aplicación.



4.3.2.3.7 View Helper (Patrón de diseño J2EE)

Intención:
Una vista delega en clases de ayuda o helpers, las responsabilidades del procesamiento del código. Estas clases o helpers encapsulan la lógica de acceso a datos y sirven como adaptadores de los datos. El patrón View Helper[11] se enfoca en recomendar formas de particionar las responsabilidades dentro de las aplicaciones.

Aplicación en Hipermedia Física:
Hay varias estrategias para implementar el componente de la vista. Una estrategia es utilizar una JSP como el componente de la vista. Y las clases de ayuda o helpers impementarlas como JavaBeans o etiquetas personalizadas. Estas tres técnica son las que recomendamos utilizar en las aplicaciones que instancien nuestro framework.



La lógica de negocio se construye fuera de las JSPs y dentro de los JavaBeans. Las etiquetas personalizadas se reutilizan, reduciendo la duplicación de código y facilitando el mantenimiento.

Encapsular la lógica de negocio en un helper en lugar de hacerlo en la vista hace que la aplicación sea más modular y facilita la reutilización de componentes. Varios clientes, como controladores y vistas, podrían utilizar el mismo helper para recuperar y adaptar estados del modelo similares para su presentación en varias vistas. La única forma de reutilizar la lógica embebida en una vista es copiando y pegando en cualquier lugar. Además, la duplicación mediante copiar-y-pegar hace que un sistema sea difícil de mantener, ya que necesitamos corregir en muchos sitios el mismo error potencial.

La vista generalmente se encuentra integrada con objetos de negocios, estos son extraídos de la misma y surgen los helpers como intermediarios entre ambos. Esto se puede observar mejor con el siguiente diagrama:

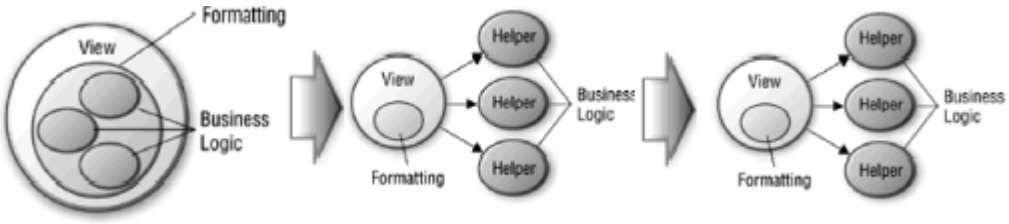


Figura 15: Extracción de la lógica de negocio en diferentes Helpers[11].

Al utilizar este patrón se logra:

- Mejorar el Particionamiento, la Reutilización y el Mantenimiento de la Aplicación. Utilizar helpers resulta en una clara separación de la vista del procesamiento de negocio.
- Mejorar la Separación de Roles. Separar el formato de la lógica de negocio reduce las dependencias que podrían existir entre los recursos.

4.3.2.3.8 Composite View (Patrón de diseño J2EE)

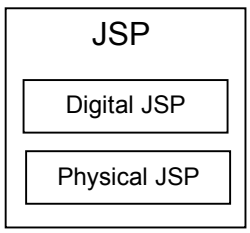
Intención:
Este patrón propone usar vistas compuestas que están formadas por múltiples subvistas atómicas, Cada componente de la plantilla puede ser incluido dinámicamente en el total, y el esquema de la página puede ser manejado de forma independiente del contenido. El patrón Composite View[11] se basa en el patrón de diseño Composite[19].

Aplicación en Hipermedia Física:
Las páginas Web presentan varios tipos diferentes de datos, utilizando varias subvistas que se completan en una sola página. Además, el desarrollo y mantenimiento de estas páginas Web, depende de muchos individuos, los cuales tienen diferentes habilidades.

Con este patrón se promueve la creación de una vista compuesta basada en la inclusión y sustitución de fragmentos modulares tanto estáticos como dinámicos. También se fomenta la reutilización de porciones atómicas de las vistas asegurando un diseño modular. Es apropiado utilizar este patrón para generar páginas que muestran componentes que podrían combinarse en una gran variedad de formas. La distribución de la página se maneja y modifica de forma independiente al contenido de las subvistas.

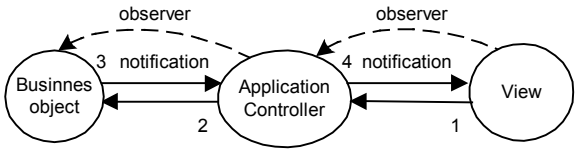
Un requisito típico en las aplicaciones Web o Wap es el que su visualización tenga una estructura similar a lo largo de toda la aplicación. Una plantilla es un componente de presentación que compone vistas separadas en una única página con un diseño específico.

Para lograr esta composición se puede utilizar en una JSP la incorporación de JSPs internas. En el caso de las aplicaciones de Hipermedia Física se tienen dos separaciones bien diferenciadas como son la parte digital y la parte física. Esto se podría representar de la siguiente manera:



4.4 Análisis de los "roles" de los componentes del framework.

En el patrón MVC, se encuentra contenido el patrón Observer. Esto se ve reflejado en la observación que hacen alguno de los componentes sobre otros. Como decidimos mantener el MVC original, estas observaciones se siguen manteniendo. Veamos esto en el siguiente diagrama:



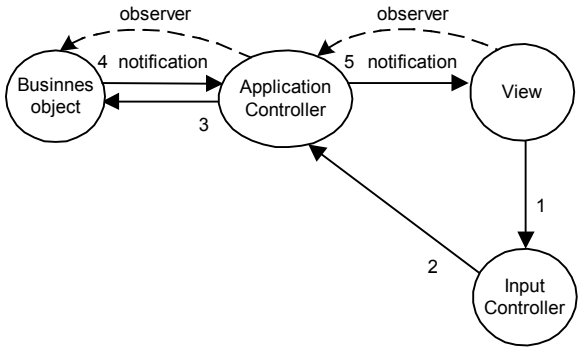
La secuencia de control es la siguiente:

1. En primera instancia la View manda un requerimiento al ApplicationController
2. El ApplicationController interactúa con el modelo para satisfacer el requerimiento recibido.
3. En el caso de producirse un cambio en el Modelo comunica del mismo al ApplicationController
4. El ApplicationController a su vez indica el cambio a la View

Este mecanismo se realiza mediante un flujo de control que se conoce comúnmente como **notification.** La notificación del cambio surge porque la View observa al ApplicationController y este a su vez observa a la parte del modelo.

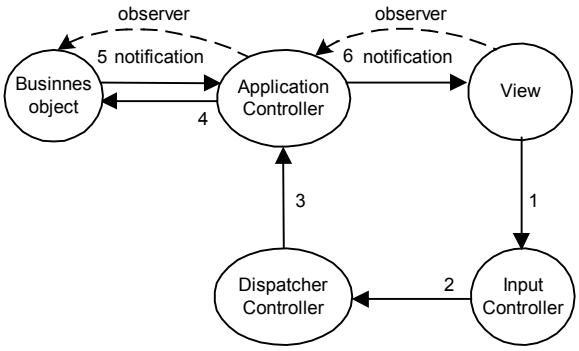
Este comportamiento se da en el MVC para Hipermedia Tradicional, y se mantiene en nuestra extensión del MVC con ubicación.

Otro comportamiento que se da en el MVC para Hipermedia Tradicional, es que el requerimiento llegue a través del InputController, como sucede en el siguiente ejemplo:



La View envía el requerimiento al InputController (1), este interactúa con el ApplicationController (2), y este último con el modelo (3). En el caso de efectuarse una modificación en el modelo se sigue la misma secuencia de control que en el diagrama donde el InputController no participaba.

En el caso de nuestra extensión este comportamiento se sigue manteniendo, pero se incorpora un componente intermedio (DispatcherController) que no modifica el esquema de observaciones y notificaciones antes descriptas. Veamos el siguiente esquema:



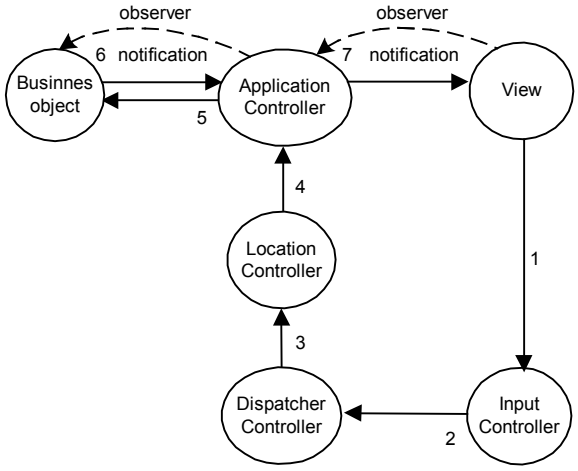
Se puede concluir que el MVC con ubicación respeta el comportamiento del MVC original. A continuación analizaremos el comportamiento que surge a partir de la incorporación de la parte física en el MVC.

Hay dos tipos de requerimientos físicos, a saber: un requerimiento explícito y un requerimiento implícito.

El requerimiento explícito lo efectúa la persona al presionar un link físico. Es decir, la persona decidió ver el camino que hay entre el lugar donde se encuentra parada hasta otro objeto físico. En este caso es la persona la que decide invocar el requerimiento.

El requerimiento implícito es el que surge cuando la aplicación "advierte" que la persona está frente a un objeto físico, en este caso la persona no realiza ninguna interacción con el sistema, sino que es la aplicación la que invoca el requerimiento.

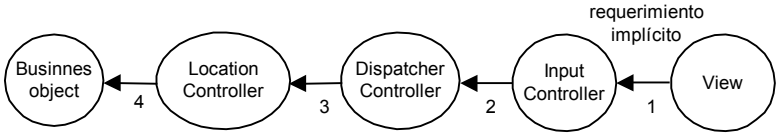
Analicemos que comportamiento tienen estos dos requerimientos en el MVC con ubicación. Primero veamos como se comporta el requerimiento explícito. Como puede observarse a continuación, debido a que es la persona quien invoca al requerimiento, este último puede verse de manera similar a un requerimiento digital:



En este caso la View se comunica con el InputController (1), este a su vez interactúa con el DispatcherController (2). Luego se pasa el control al LocationController (3) y este deriva el control al ApplicationController una vez que procesó la parte de ubicación. De aquí en adelante el flujo de control se realiza de la misma manera que lo haría un requerimiento que no involucre ubicación.

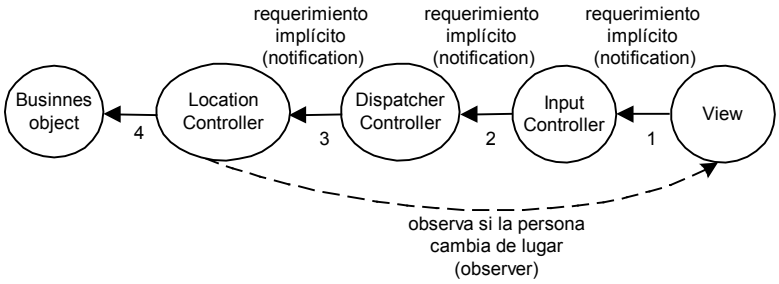
El ApplicationController no conoce la naturaleza del requerimiento. Es decir, desconoce si el requerimiento es digital o es un requerimiento fisco explícito.

El requerimiento implícito surge a partir de que la persona interactúa en el mundo real. Al moverse y pasar frente a los distintos objetos físicos, se genera este nuevo comportamiento. Veamos que componentes se involucran en un requerimiento implícito para analizar como se comportan. Para esto podemos ver el siguiente diagrama:



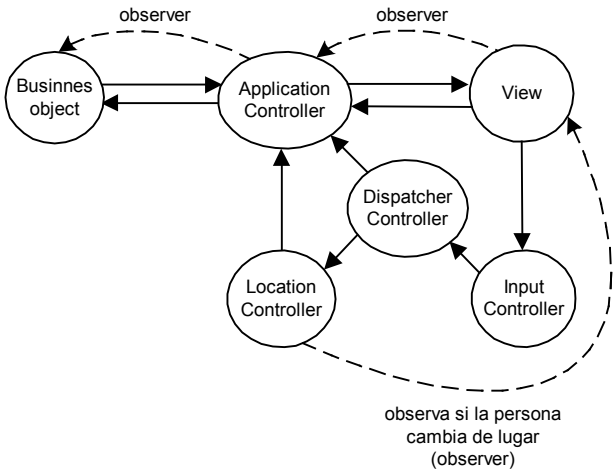
La View "advierte" mediante algún mecanismo que la persona se encuentra parada frente a un determinado objeto físico. Esto origina un requerimiento implícito al InputController (1), este pasa el control al DispatcherController (2). Luego se cede el control al LocationController (3), el cual interactúa con el modelo (4). De esta manera el LocationController busca en el modelo el objeto físico que se encuentra frente a la persona.

El LocationController se podría ver como un observador de la View, y el requerimiento implícito se podría ver como la notificación del cambio de la posición de la persona. Es decir, estamos nuevamente en presencia del patrón Observer. Veamos la siguiente representación:



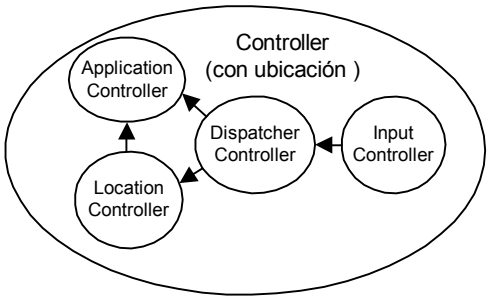
Es aquí donde surge un comportamiento específico propio de la relación de la aplicación con el mundo físico.

Una manera de mostrar en forma conjunta tanto el comportamiento físico como el digital, es el siguiente:

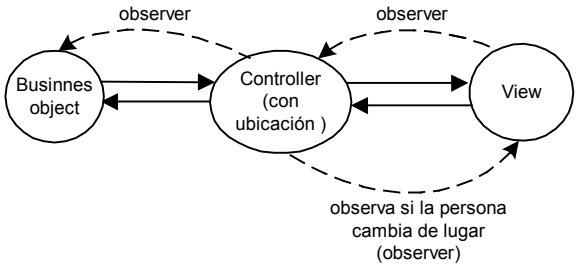


Se puede concluir que la incorporación de la ubicación al MVC genera que la Vista tenga como "roles" ser observador del Modelo (a través de observar al ApplicationController) y ser observado por el LocationController. Es decir, se le agrega un "nuevo rol" al que tenía en el MVC tradicional. Originariamente la vista sólo actuaba como observador del modelo, luego de este análisis podemos concluir que también será observada por el LocationController.

Veamos que "roles" se le atribuyen al controlador que surge de la extensión del MVC. Todos los componentes del controlador se pueden encapsular de la siguiente manera:



Si se reemplaza en el diagrama el controlador con esta abstracción quedaría de la siguiente manera:



Este esquema permite apreciar que el Controlador (con ubicación) tiene los siguientes "roles": es observador de la Vista y del Modelo, y es observado por la Vista. Por lo tanto el "rol" que se incorpora al Controlador además de los del MVC tradicional, es el de ser observador de la Vista. Esto es por el comportamiento que surge de la parte física.

En el caso del modelo no incorpora nuevos "roles", sino que mantiene el del MVC tradicional, que es ser observado por el controlador.

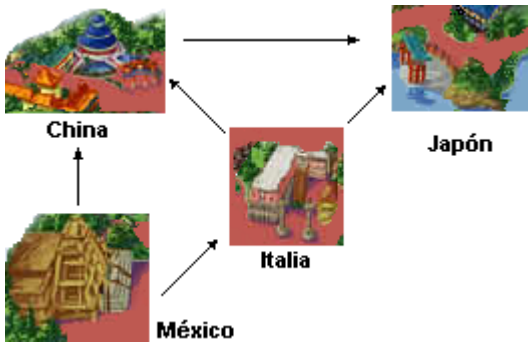
En conclusión, la Vista y el Controlador (con ubicación) cambian el comportamiento al tener incorporada la parte de la movilidad del usuario. Como la Vista tiene que hacer un requerimiento implícito al Controlador se genera un cambio fundamental en el comportamiento de los componentes del MVC con ubicación. Esto es esencial a la hora de desarrollar alguna aplicación de Hipermedia Física, la cual involucre ubicación del usuario. En el caso de estas aplicaciones no solamente cambia el modelo, sino que es la ubicación del usuario lo que sufre una alteración. Es decir, que las vistas ya no dependen sólo de un modelo, sino que dependen además de la posición del usuario.

Capítulo 5 Ejemplo-caso de Uso

5.1 Dominio del Ejemplo

Este capítulo, especifica los pasos para instanciar el StrutsLocationFramework en el contexto de un parque temático. En especial optamos por simular el World Showcase de Disney. Este parque ofrece como atractivo viajar por diferentes culturas sobresalientes del mundo.

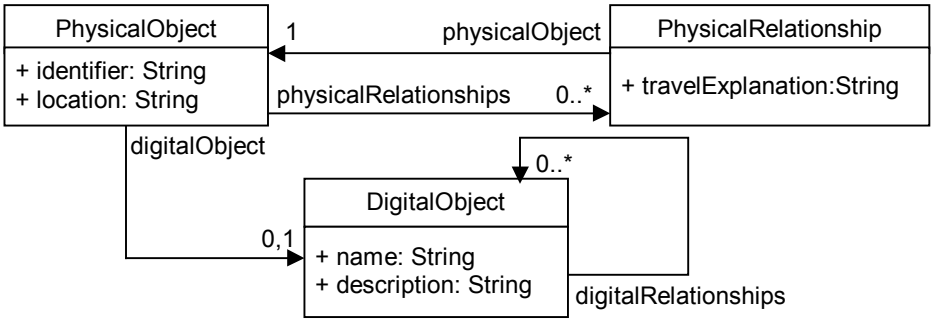
Si bien este parque temático ofrece la opción de conocer la cultura de 11 naciones, el ejemplo que citaremos a continuación se restringirá a 4 de ellas. En particular proponemos un escenario como el que muestra la siguiente figura:



Optamos por elegir un atractivo de cada país y también restringimos las opciones de caminos entre ellos.

5.2 Modelado de los objetos físicos y digitales

En general, para este tipo de aplicaciones se necesitan como mínimo los siguientes objetos y relaciones que modelamos a continuación:

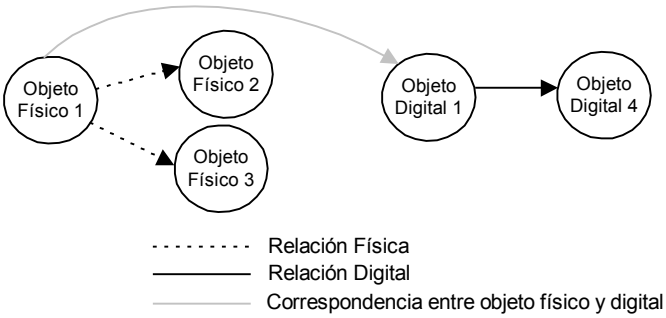


Los objetos digitales serán representados con una abstracción de sus características principales, por ejemplo: **name**, **description** y las relaciones que tienen con otros objetos digitales. Como se ve en el diagrama anterior la variable **digitalRelationships** representa las relaciones que tiene el objeto digital con otros objetos digitales. Esto permitirá la navegación digital dentro de la aplicación.

En el caso de los objetos físicos tendrán como característica genérica, un **identifier** unívoco según el sistema de posicionamiento elegido para la aplicación. Por simplicidad en este ejemplo alcanza con tener un String que identifique al objeto en el sistema, y optamos por asignar números para este fin. También tendrá un **location** "textual" de su ubicación relativa al mundo real, y las relaciones con otros objetos físicos (**physicalRelationships**). Es importante notar que estas relaciones nacen de la necesidad de poder interconectar los objetos físicos. La clase **PhysicalRelationship** intenta modelar en este caso particular el recorrido que debe realizar la persona para llegar de un objeto a otro. Para acotar el dominio y simplificar nuestra tarea elegimos un String que especifique el nombre del mapa que permite a la persona llegar al destino buscado.

Como es de esperar estas clases se podrían subclasificar para lograr especificar más nivel de detalles de los objetos o tener en la aplicación un comportamiento más especifico de los mismos. Por simplicidad, representaremos cada atracción del parque como un objeto físico. Cabe aclarar que dentro de una atracción podrían existir a su vez nuevos objetos físicos relevantes y de esta manera seguir aumentando la complejidad del ejemplo tanto como se desee.

Volviendo al diagrama, se puede apreciar que un objeto físico puede conocer o no, a un objeto digital. Esto quiere decir que un objeto puede estar representado en el mundo real (mediante un objeto físico) y digital. Esta relación de conocimiento permite asociar a los objetos físicos toda la información digital que se tenga de dicho objeto. De esta manera se logra mostrar información, no precisamente física, a la persona que se encuentra frente a un objeto físico. Pueden existir objetos físicos que no tengan una asociación con información digital, ya que puede considerarse innecesaria para la aplicación. También se puede dar el caso inverso, y que un objeto digital no tenga una representación en el mundo real. Lo antes mencionado se puede observar en el siguiente diagrama:



El **Objeto Físico 1** se relaciona físicamente con el **Objeto Físico 2** y el **Objeto Físico 3**. Su parte digital está representada por el **Objeto Digital 1**. Este a su vez se relaciona digitalmente con el **Objeto Digital 4.**

En nuestro ejemplo, cada atracción del parque tendrá un objeto digital, que especificará sólo el nombre y la descripción, si bien esto podría extenderse tanto como uno lo desee.

5.3 Instanciación del Framework

Se detallarán a continuación los pasos que deben seguirse para tener el ejemplo funcionando con el Framework propuesto.

I. Agregar el Framework como una librería. (StrutsLocationFramework.jar)

II. Crear el buscador. En este ejemplo se denomina *ExampleLocationFinder* e implementa los siguientes métodos:

```
public Object inFrontOfObject(HttpServletRequest req)

public Object howToReachFrom(Object from, HttpServletRequest to)
```

El primero sirve para buscar el objeto frente a la persona, y el segundo para buscar el camino entre dos objetos.

En nuestro ejemplo cada atracción (u objeto físico) se identifica con un número.

1 → Pirámide mexicana (México)

2 → Templo del Cielo (Pekín, China)

3 → El Palacio del Dogo (Venecia, Italia)

4 → La Puerta Tori (Japón)

Por lo tanto se implementa el método inFrontOfObject(HttpServletRequest req) de manera que dependiendo el valor del parámetro id guardado en el request devuelva uno de los 4 objetos físicos.

De la misma manera se implementará el método howToReachFrom(Object from, HttpServletRequest to) para que devuelva un mapa especificando como llegar de una atracción a otra.

III. Configurar el archivo web.xml. En nuestro ejemplo dicho archivo queda configurado de la siguiente manera:

```
<web-app>
...
   <init-param>
         <param-name>finder</param-name>
         <param-value>example.finder.ExampleLocationFinder</param-value>
   </init-param>
   <init-param>
         <param-name>objectName</param-name>
         <param-value>physicalObject </param-value>
   </init-param>
   <init-param>
         <param-name>travelName</param-name>
         <param-value>physicalTravel</param-value>
   </init-param>
…
</web-app>
```

Cuando se especifica **example**.**finder.ExampleLocationFinder,** significa que en el paquete **example** (donde está especificado nuestro ejemplo), hay un paquete **finder** que contiene la clase **ExampleLocationFinder.**

IV. En el archivo de configuración struts-config.xml, especificar los *Action* relacionados con la parte digital. Para nuestro ejemplo sólo tenemos que buscar objetos digitales. Es decir, nos basta con tener un *Action* para las búsquedas digitales. Esto quedaría especificado en el archivo de la siguiente manera:

```
<struts-config>
...
      <action-mappings>
      ...
            <action  path="/DigitalRelationship"
                        type="example.actions.DigitalRelationshipAction"/>
       ...
       </action-mappings>
...
</struts-config>
```

El atributo **path** permite especificar como será invocado en las JSP este *Action.* En el atributo **type** se detalla la clase del *Action* que atenderá el pedido, en este caso será **DigitalRelationshipAction** que se encuentra en el paquete **actions,** el cual está contenido en el paquete **example.**

V. En el archivo de configuración location-struts-config.xml, especificar la parte física. Para nuestro ejemplo esto quedará de la siguiente manera:

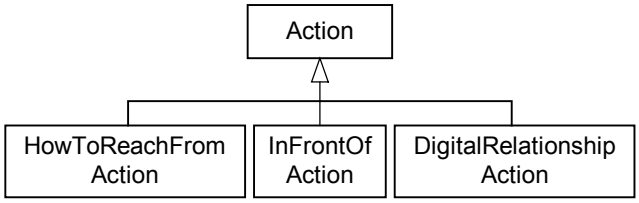
```
<struts-config>
...
      <location-action-mappings>
      ...
         <location-action      path="/InFrontOf "
```

```
                    locationEvent=" inFrontOf "
                     type="example.locationActions.InFrontOfAction"/>

    <location-action    path="/HowToReachFrom"
                        locationEvent="howToReachFrom"
                        type=" example.locationActions.HowToReachFromAction"/>
    ...
  </location-action-mappings>
...
</struts-config>
```

Se puede observar que el paquete **example,** contiene otro paquete llamado **locationActions** en el cual están las clases **InFrontOfAction** y **HowToReachFromAction**. La primera será para atender los pedidos que surjan cuando una persona está frente a un objeto fisco. Y la segunda cuando la persona decide caminar desde un objeto físico a otro.

VI. Definir las clases *InFrontOfAction*, *HowToReachFromAction* y *DigitalRelationshipAction*, las cuales deben ser subclases de la clase *Action***.**



VII. Luego queda definir las JSP según los datos que se quieran mostrar.

Una vez completados estos pasos el ejemplo queda listo para ser utilizado.

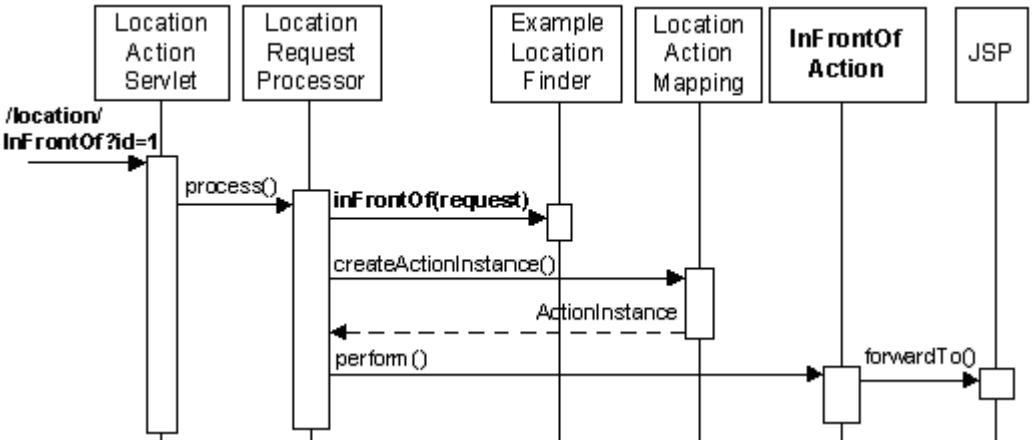
5.4 Secuencia de pasos que se disparan cuando la persona está frente a un objeto físico

Cuando la persona pasa frente a un objeto físico, el censor manda una señal al dispositivo móvil. El mismo genera un requerimiento, el cual tendrá la siguiente forma:

```
/location/InFrontOf?id=1
```

Cuando le llega el requerimiento, el framework detecta que la URL se corresponde con la parte de ubicación, y por lo tanto cede el control al *LocationActionServlet*. Este delega en el *LocationRequestProcessor* la resolución del pre-procesamiento del requerimiento. El *LocationRequestProcessor* invoca en el *ExampleLocationFinder* el método inFrontOf(HttpServletRequest req). En este método se buscará al objeto físico que se identifique con el número 1. El *ExampleLocationFinder* detecta el objeto físico correspondiente y lo guarda en la sesión bajo el nombre physicalObject.

Una vez que se realizó el pre-procesamiento del requerimiento, el control vuelve al *LocationRequestProcessor* quien consulta a los *LocationActionMapping* para detectar que *Action* se corresponde con la URL en cuestión. Según la configuración del location-struts-config.xml, el que debe atender el requerimiento es el *InFrontOfAction.* Luego se genera la JSP correspondiente. Esto puede quedar representado con el siguiente diagrama:



Para entender mejor esta situación supongamos que la persona llega a la Pirámide Mexicana. Por algún mecanismo (por ejemplo por algún censor) se enviará el requerimiento:

```
/location/InFrontOf?id=1
```

Que tendrá como resultado en la pantalla del dispositivo de la persona una visualización similar a la que se muestra a continuación:

En este momento la persona tiene la posibilidad de ver más información digital mediante el link México. La otra opción es que le interese por ejemplo conocer El Templo del Cielo de Pekín (China). Mediante el link físico Templo del Cielo se accederá a información que le permitirá a la persona caminar hasta el mismo. Dicho requerimiento se explica a continuación.

5.5 Secuencia de pasos que se disparan cuando la persona decide caminar de un objeto físico a otro

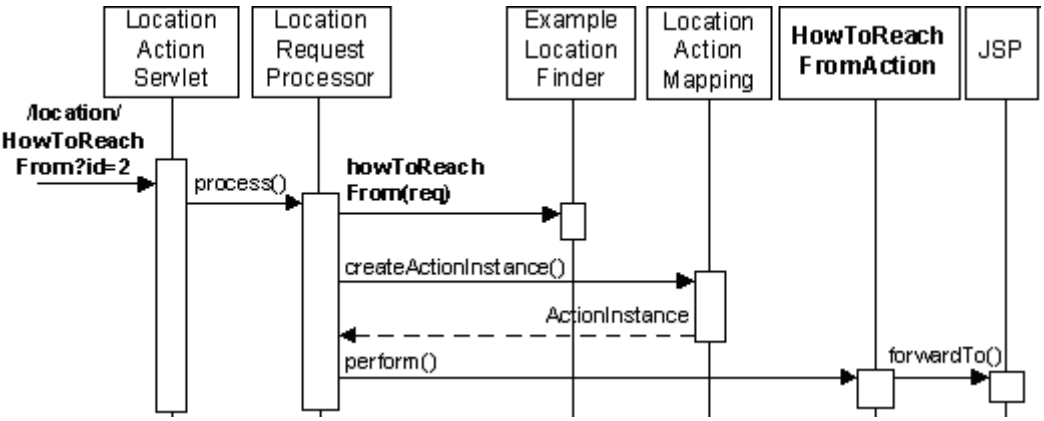
Este requerimiento difiere del anterior, en que no es disparado por el sistema, sino que surge cuando la persona presiona un link físico. La URL que representa el requerimiento que está en el link tiene la siguiente forma:

```
/location/HowToReachFrom?id=2
```

El flujo de control es el mismo que en el caso anterior, salvo que el *LocationRequestProcessor* invoca en el *ExampleLocationFinder* el método howToReachFrom(HttpServletRequest request). En este método buscará el camino entre el objeto que está frente a la persona y el objeto que se hace

referencia en el link (en este caso el que se identifica con el 2). El *ExampleLocationFinder* detecta el camino físico, que es guardado en la sesión bajo el nombre physicalTravel.

Como en el caso anterior, una vez que se realizó el pre-procesamiento del requerimiento, el control vuelve al *LocationRequestProcessor* quien consulta a los *LocationActionMapping* para detectar que *Action* se corresponde con la URL en cuestión. En este caso la configuración del archivo location-struts-config.xml, indica que debe ser atendido por el *HowToReachFromAction*. Luego se generará la JSP mostrando como llegar de un objeto al otro. El flujo de control puede representarse de la siguiente manera:



En nuestro ejemplo si la persona se encontraba en la Pirámide Mexicana y presionó el link físico para llegar al Templo del Cielo, en su dispositivo móvil verá un mapa que le muestra el recorrido que lo guiará para realizar el trayecto, como se puede apreciar en el siguiente dibujo:

Notar que esto puede ser tan complejo como uno lo desee o tan simple como una descripción textual. Una alternativa sería mostrar un mapa que muestre dicha trayectoria con señales que le ayuden a la persona a ubicarse mientras realiza el recorrido, o podría acompañarse de un audio o video con la explicación.

Supongamos que la persona realiza el recorrido y una vez que tiene enfrente el Templo del Cielo se le muestra la siguiente pantalla generada a través del requerimiento que impulsó la señal del censor.

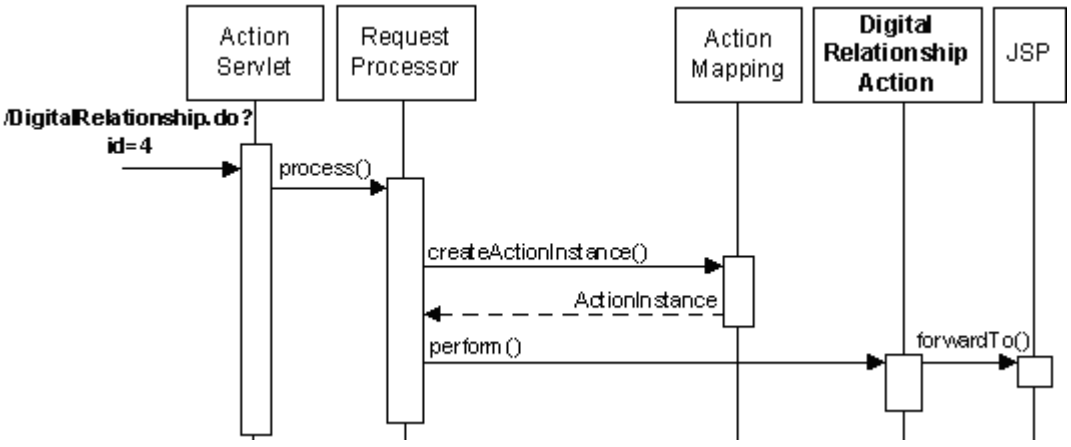
Si en dicha pantalla se presiona el link digital que solicita información sobre China, se realizarán los pasos que se detallan en el siguiente punto.

5.6 Secuencia de pasos que se disparan cuando la persona genera un requerimiento digital

Este tipo de requerimiento se genera cuando la persona clickea un link digital que tiene asociada una URL con la siguiente forma:

/DigitalRelationship.do?id="4"

Cuando le llega el requerimiento, el framework detecta que la URL se corresponde con la parte digital, y cede el control al *ActionServlet.* Este delega en el *RequestProcessor* la resolución del requerimiento. El cual consulta al los *ActionMapping* para detectar que *Action* se corresponde con la URL en cuestión. Según la configuración del struts-config.xml, el que debe atender el requerimiento es el *DigitalRelationshipAction.* Luego se genera la JSP correspondiente. Esto puede observarse en el siguiente diagrama:



En nuestro ejemplo, al solicitar información sobre China, podría aparecer una pantalla similar a la siguiente:

Vale aclarar que esta pantalla mantiene los mismos links físicos que la anterior, debido a que la persona todavía se encuentra en el mismo sitio.

5.7 Pruebas en diferentes Browsers

Una vez que se especificaron todos los pasos y el flujo de control que se ejecuta para cada pedido del ejemplo, es oportuno destacar que este puede ser utilizado en diferentes Browsers.

Para llevar a cabo la simulación que demuestra que la instanciación es apta para diferentes tipos de Browsers, usamos un simulador de Palm y uno de teléfonos móviles. Estos permiten la simulación Wap, necesaria para la naturaleza de las aplicaciones de Hipermedia Física. La parte que varía para cada una de las simulaciones es exclusivamente las JSPs. Estas constarán con tags wml. También utilizamos un Browser Web para ratificar que las aplicaciones se pueden visualizar en el mismo.

Para la simulación de que la persona se encuentra frente a la "Pirámide Mexicana", en cada uno de los Browser se visualizaría lo siguiente:

Se puede observar en todos los casos la misma información, distribuida según las capacidades de cada uno de los dispositivos o Browsers. Aclaremos que conocer las características de los dispositivos móviles es fundamental para lograr una buena visualización de los datos.

Si en la simulación se elige caminar hasta el "Templo del Cielo", en cada uno de los Browsers se visualizaría lo siguiente:

De esta manera se puede ver la simulación de una aplicación de Hipermedia Física, en diferentes Browsers tanto sea Web como Wap instanciando nuestro framework.

Capítulo 6 Comparaciones

6.1 Trabajos relacionados sobre Hipermedia Física

6.1.1 GeoNotes

El sistema GeoNotes[15] intenta unir áreas bastante dispares de investigación dentro de la interacción hombre-máquina. Intenta borrar el límite con el espacio físico y al mismo tiempo reforzar el espacio digital socialmente.

Basado en tecnología de posicionamiento permite añadir notas virtuales a través de algún dispositivo móvil a ubicaciones del mundo real. Cuando otra persona pasa por este sitio le llegará a su dispositivo móvil una notificación informándole de la existencia de información, y luego la persona podrá leerla.

Los usuarios, de esta manera, podrán observar y participar en los modelos sociales del espacio geográfico. GeoNotes crea un awareness social que incentiva la expresión y formación de la identidad personal.

GeoNotes utiliza GPS y DGPS, esto es útil solo para áreas físicas muy grandes. El problema que tienen estas tecnologías es que no son exactas, poseen un margen de error en el cálculo de la posición y raramente se utilizan en edificios o ambientes interiores.

6.1.2 HyCon

HyCon[26] es un framework que extiende el paradigma de Hipermedia con el mundo físico. Sirve para sistemas de Hipermedia context-aware. Lo interesante de este sistema es que soporta los mecanismos clásicos de Hipermedia para navegar, buscar, hacer notaciones y tours guiados en el mundo físico, permitiendo a los usuarios realmente unir objetos digitales y/o físicos.

Su arquitectura da soporte para realizar anotaciones, manejo de links y tours guiados asociando ubicaciones y objetos con dispositivos RFID o Bluetooth con mapas, páginas web y colecciones de recursos. Cabe destacar también que esta arquitectura propuesta, incluye interfaces para incluir la capa de censores, que encapsulan el GPS y otros sistemas de posicionamiento que se pueden utilizar. Introduce el uso de XLinks y un nuevo sistema de búsqueda para la web llamado Geo-Based Search (GBS).

Los objetivos de HyCon son: extender Hipermedia con el mundo físico, soportar colecciones automáticas de información contextual e información social, y funcionar con dispositivos móviles heterogéneos.

6.1.3 HyperReal

HyperReal[43] es un framework orientado a objetos que permite construir aplicaciones context-aware y mixed reality. Para lograr dicho framework se establece un modelo genérico que integra conceptos ya introducidos en este área, basándose en los conceptos de Dexter establecidos para el Hipertexto. Además para incorporar los diferentes medios, utiliza los conceptos de Hipermedia adaptativa y espacial. Integra en el mismo framework documentos virtuales, ambientes 3D y el mundo físico para construir aplicaciones de mixed reality.

El modelo genérico especifica una base estructural para establecer la relación entre los espacios reales o virtuales y los soportes de mecanismos contextuales. La representación del espacio virtual y físico está integrada en el modelo, permitiendo modelar las relaciones entre los diferentes tipos de elementos que están ubicados en el espacio.
HyperReal puede manejar diferentes elementos, y define un esquema de presentación que abstrae los conceptos de navegación relevantes, incluyendo el awareness de los links. El awareness de los links provee una manera de representar la información de navegación. En los ambientes de mixed reality, esto es particularmente importante para tener este mecanismo en el nivel del modelo, dando un dinamismo natural a la representación del link.

6.1.4 ProXimity

El proyecto proXimity[27] es un sistema que busca aumentar la realidad dando al Hipertexto, una presencia física en el mundo verdadero. Está basado en trabajos sobre Hipermedia y movilidad en el mundo real e intenta extender la metáfora del link de Hipermedia en el espacio físico.

En este proyecto surge el concepto de " walk the Links" [27] o *"caminar los links"*. Este trabajo establece fuertes indicios para suponer que en el futuro, el Hipertexto y el mundo real van a unirse y permitirán que el usuario en efecto "camine los links".

El principal aporte de proXimity es establecer tres componentes para el Hipertexto en la Web. A saber un componente espacial, uno temporal y otro semántico. El componente semántico permitirá agrupar los links bajo algún criterio. El temporal dará la información de la ubicación tanto de la persona como de los objetos del mundo real, y el espacial resolverá las cuestiones de como atravesar las distancias.

ProXimity es eficaz en el uso de nuevas tecnologías sobre el conocimiento, ambientes y dominios del usuario (semántica Web, IR, Bluetooth, GPRS), esto lo convierte en un sistema único y diferente.

6.1.5 TOPOS

TOPOS [24], [25] es un prototipo que permite la manipulación y mantenimiento de relaciones espaciales entre materiales en un ambiente tridimensional. Se integra con aplicaciones para soportar colaboración en tiempo real. Esta es una manera de aumentar la realidad multidimencionalmente.

El objetivo fue desarrollar un sistema de Hipermedia Física para dar a los usuarios, un ambiente familiar al mundo real en el que trabajan. Organizando las mezclas entre materiales digitales y físicos, mediante colecciones de objetos. El concepto central en Topos es el *workspace* que es el medio principal para agrupar y organizar los materiales y objetos en el espacio 3D. Otro aspecto interesante que provee es que permite agrupar, mezclar y conectar los espacios de trabajo de varias maneras diferentes.

Se puede utilizar tanto para sistemas de Hipermedia espacial abstractos con espacios abiertos o concretos usando por ejemplo un edificio como un "background" para la colocación relativa de materiales.

Se integra un sistema de tag RFID para soportar la creación, manipulación y mantenimiento de las relaciones en el ambiente interactivo. RFID se utiliza para registrar material físico y generar un ID digital.

6.2 Comparación con los Trabajos relacionados

Hay que diferenciar la naturaleza de cada uno de los trabajos, entre los mismos se encuentran un prototipo (TOPOS), un sistema (GeoNotes), un proyecto (ProXimity) y dos frameworks (HyCon y HyperReal).

Nuestro trabajo tiene una relación más profunda con los dos framework (HyCon y HyperReal). Si bien nuestro framework y los antes mencionados se basan en Hipermedia para extender el espacio físico, en particular nuestro trabajo se diferencia en que está basado en el concepto de MVC. Esto es una ventaja para nuestro framework ya que trae aparejado como beneficio disminuir las dependencias. El modelo MVC permite una separación casi perfecta entre lo que se conoce como modelo (ó la lógica de negocio), el controlador y la vista.

Cualquier aplicación que se desarrolle bajo los frameworks HyCon o HyperReal, puede ser desarrollada por nuestro framework. Considerando las tecnologías que se adecuen más a cada una de las aplicaciones.

En cuanto al concepto de ProXimity de "caminar el link", se corresponde con los links físicos de nuestro framework. Cuando una persona presiona un link físico, se le brinda la información necesaria para que logre "caminar" desde donde se encuentra hasta el lugar que representa el link físico.

GeoNotes también podría ser implementado en una aplicación que utiliza nuestro framework. Hay que destacar en este punto que los comentarios podrían estar relacionados con ambientes físicos en vez de objetos físicos. En vez de esperar a que una persona se encuentre frente a un objeto en particular para mostrarle información, cuando la persona ingresa a un área en particular se le mostrará información que otros guardaron del lugar. Hay que considerar la posibilidad de guardar comentarios de la persona, los cuales en la navegacion física a veces nos son considerados.

TOPOS posee dos ramas separadas de desarrollo, una sobre aplicaciones de escritorio y otras relacionada con los ambientes físicos. Nuestro framework abarca solamente la parte de aplicaciones con ambientes físicos. Las aplicaciones de escritorio que ellos desarrollan están fuertemente orientadas a soportar manipulación y organización de materiales físicos a través de representaciones digitales, función que no se desarrolla en nuestro trabajo.

En resumen nuestro framework podría instanciarse para cualquier aplicación de Hipermedia Física, basada en Hipermedia, teniendo en cuenta para las aplicaciones el concepto de "caminar el link".

Capítulo 7 Conclusiones

Recordemos objetivos que nos habíamos propuesto para esta tesis:

- Principalmente la extensión del concepto MVC para aplicaciones de Hipermedia Física.
- En segundo término descripciones de patrones de diseño que se puedan utilizar en aplicaciones de Hipermedia Física.
- Por último un análisis de los "roles" que se desprenden de cada uno de los componentes de la extensión del MVC en una aplicación de Hipermedia Física.

7.1 Extensión del concepto MVC para aplicaciones de Hipermedia Física

Al concluir esta tesis podemos decir que hemos encontrado un framework para aplicaciones de Hipermedia Física, que simplifique el trabajo de los desarrolladores, disminuyendo el esfuerzo necesario para construirlas.

El framework se desarrolló como una extensión del MVC que permite soportar ubicación. Para lograr implementarlo se utilizó Struts, como base para llevar a cabo la extensión. Primero realizamos una extensión conceptual del MVC, y luego se implementó dicho diseño logrando el framework en cuestión.

Una consideración importante fue lograr mantener intacta la implementación existente de Struts, que nos sirvió para manejar los requerimientos tradicionales. Se incorporaron al framework de Struts, elementos propios para la parte de ubicación a saber: *LocationActionServlet*, *LocationRequestProcessor, LocationActionMapping, LocationFinder, LocationConfigRuleSet.* Estos

permitieron manejar los requerimientos relacionados con ubicación. Para poder manejarlos principalmente se incorporó un pre-procesamiento para lograr ubicar los objetos físicos o el camino físico.

Con el desarrollo del framework se introdujo, además una librería de tags relacionada a las aplicaciones de Hipermedia Física. Dicha librería añade la posibilidad de trabajar con objetos que no sean sólo beans. Esto le permite a los desarrolladores realizar por ejemplo filtrados en la información que brinden los objetos físicos.

Algo que queremos destacar es que el trabajo es independiente del tipo de Browser que se utilice para la visualización de la aplicación. Es decir, puede utilizarse tanto para Browsers Web, como Browsers Wap. En el caso de aplicaciones de Hipermedia Física que se relacionan con ubicación es fundamental que el framework funcione con Browsers Wap, por la naturaleza de este tipo de aplicaciones. Es decir, como se necesitan dispositivos móviles, para que el usuario se traslade con ellos, resulta imprescindible que el framework soporte este tipo de Browsers. En el caso de los Browsers Web, las aplicaciones de Hipermedia Física serían apreciadas como aplicaciones de Hipermedia Tradicional.

Otra ventaja del framework, es que resulta muy sencillo de instanciar. Sólo se necesita:

- Especificar un buscador (para encontrar objetos físicos y el camino entre dos objetos físicos), el nombre con que se guardarán en la sesión los objetos físicos y el camino físico.
- Crear los *Action* de la aplicación y configurarlos en los archivos correspondientes dependiendo si están relacionados con ubicación o no (location-struts-config.xml y struts-config.xml).
- Y crear las JSPs dependiendo de la información que se quiera mostrar.

7.2 Patrones de diseño que se pueden utilizar en aplicaciones de Hipermedia Física

Con este trabajo logramos alcanzar la especificación buscada de los patrones de diseño que pueden utilizarse en este tipo de aplicaciones. Los patrones encontrados según el libro "Design Patterns: Elements of Reusable Object Oriented Sofware"[19] fueron: Builder, Composite, Decorator, Observer, Strategy y Template Method.

Los beneficios que origina el uso de patrones pueden ser medidos en varios sentidos:

- Contribuyen a reutilizar diseño, identificando aspectos claves de la estructura de un diseño que puede ser aplicado en una gran cantidad de situaciones.
- Mejoran (aumentan, elevan) la flexibilidad, modularidad y extensibilidad.
- Ayudan a diseñar desde un mayor nivel de abstracción, aumentando nuestro vocabulario.

Existe además un conjunto de patrones que pueden ser usados en el contexto del diseño de aplicaciones Java 2 Enterprise Edition, J2EE. Los patrones de diseño J2EE consisten en soluciones recurrentes y documentadas de problemas comunes de diseño de aplicaciones J2EE. Según los patrones J2EE especificados en “J2EE PATTERNS Best Practices and Design Strategies”[11] encontramos que se pueden aplicar el View Helper y el Composite View a las aplicaciones de Hipermedia Física.

7.3 Los “roles” de los componentes de la extensión del MVC

Por último realizamos una descripción de los “roles” intervinientes en nuestra extensión del MVC con ubicación. La Vista y el Controlador (con ubicación) cambian el comportamiento al tener incorporada la parte de la movilidad del usuario.

El Modelo no incorpora nuevos “roles”, si no que mantiene el del MVC tradicional, que consiste en ser observado por el controlador.

El Controlador (con ubicación) tiene los siguientes "roles": es observador de la Vista y del modelo, y es observado por la propia Vista. Por lo tanto el "rol" que se incorpora al Controlador además de los del MVC tradicional, es el de ser observador de la Vista. Esto es por el comportamiento que surge de la parte física.

La incorporación de la ubicación al MVC genera que la Vista tenga como "roles" ser observador del modelo y ser observado por el LocationController. Es decir, se le incorpora un "rol" más al que tenía en el MVC tradicional, que era sólo ser observador del modelo, ser observada por el LocationController.

Como la Vista tiene que hacer un requerimiento implícito al Controlador se genera un cambio fundamental en el comportamiento de los componentes del MVC con ubicación. Esto es esencial a la hora de desarrollar alguna aplicación de Hipermedia Física, la cual involucre ubicación del usuario. En el caso de estas aplicaciones no solamente cambia el modelo, sino que es la ubicación del usuario lo que sufre una alteración. Es decir, que las vistas ya no dependen sólo de un modelo, sino que dependen además de la posición del usuario.

Capítulo 8 Referencias bibliográficas

[1] J. Antoniucci: "Introducción a Struts".

[2] R.T. Azuma: "A Survey of Augmented Reality". Presence: Teleoperators and Virtual Environments.vol.6, no.4, August 1997,pp.355-385.

[3] R.T. Azuma, Y. Baillot, R. Behringer, St. Feiner, S. Julier, B. MacIntyre: "Recent Advances in Augmented Reality". IEEE, Nov. 2001, pp 255-385.

[4] R. Balzer, M. Begeman, P. Garg, M. Schwartz, B. Shneiderman: "Hypertext and Software Engineering". Hypertext 1989. pp.395-396.

[5] N.O. Bouvin, A. Universitet: "Spatial hypermedia 1.0". 29/Apr/2004. www.daimi.au.dk/~bouvin/hypermedier/2004/10/talk.html.

[6] F. Buschman, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: "A System of Patterns". John Wiley and Sons, New York, 1996.

[7] J. Cabero Almenara: "Navegando, construyendo: la utilización de los Hipertextos en la enseñanza". Medios de comunicación, recursos y materiales para la mejora educativa II, Sevilla, CMIDE del Excmo. Ayuntamiento de Sevilla y SAV de la Universidad de Sevilla, 227-243. http://tecnologiaedu.us.es/revistaslibros/hiper.html.

[8] S.A. Carter, E. Churchill, L. Denoue, J.I. Helfman, L. Nelson: “Digital Graffiti: Public Annotation of Multimedia Content”. CHI 2004, Vienna, Austria, April 24-29, 2004. New York: ACM Publications. February 26, 2004

[9] J. Conklin: "Hypertext: An Introduction and Survey". IEEE Computer, Septiembre 1987. pp. 17-41

[10] J.O. Coplien, D.C Schmidt: “Pattern Languages of Program Design”. Addison-Wesley, 1995.

[11] J. Crupi, D. Malks, D. Alur: “J2EE PATTERNS Best Practices and Design Strategies”. Publisher: Prentice Hall / Sun Microsystems Press, June 2001

[12] A. Dabkowski, A.M. Jankowska, K. Kurbel: “Model-View-Controller Design Pattern for Mobile and Desktop-based Applications”. Proceedings of the 8th International Workshop on Mobile Multimedia Communications, Munich, 2003, pp. 467-471.

[13] N. Dai, A. Knight: “Objects versus the Web”. OOPSLA 2001 tutorial; www.smalltalkchronicles.net/papers/objectsXweb10152001.pdf.

[14] M. Ellis, N. Dai: “Best Practices for Developing Web Applications Using Java Servlets”. OOPSLA 2000 tutorial; www.smalltakchronicles.net/papers/Practices.pdf.

[15] F. Espinoza, P. Persson, A. Sandin, H. Nystrom, E. Cacciatore, M. Bylund: “GeoNotes: Social and Navigational Aspects of Location-Based Information Systems”. Proceedings of Third International Conference on Ubiquitous Computing (Ubicomp 2001), pp 2-17. Springer Verlag.

[16] D. Estrin, D. Culler, K. Pister, G. Sukhatme: “Connecting the Physical World with Pervasive Networks”. IEEE Pervasive Computing, vol. 01, no. 1, pp. 59-69, January-March 2002.

[17] J. Falker, K. Jones: “Servlets and JavaServer Pages™: The J2EE technology Web tier”. Publisher: Addison-Wesley Published 2003/09

[18] J. Fiderio: "A Grand Vision--Hypertext mimics the brain's ability to access information quickly and intuitively by reference". Byte Magazine, Vol. 13, Nº 10. October 1988. pp.237—244.

[19] E. Gamma, R. Helm, J. Johnson, J. Vlissides: “Design Patterns. Elements of reusable object-oriented software”. Addison Wesley 1995.

[20] J.M. Garnier: “MVC Model 2 and Struts 1.1”. http://rollerjm.free.fr/pro/Struts11.html

[21] J. Goodwill: “Mastering Jakarta Struts. Chapter 1: Introducing the Jakarta Struts Project and Its Supporting Components”. Wrox Press, Paperback, Published September 2003.

[22] S. Gordillo, G. Rossi, F. Lyardet: “Modeling Physical Hypermedia Applications”. SAINT Workshops 2005: 410-413

[23] S. Gordillo, G. Rossi, D. Schwabe: “Separation of Structural Concerns in Physical Hypermedia Models”. CAiSE 2005: 446-459

[24] K. Grønbæk, J.F. Kristensen, P. Ørbæk, M.A. Eriksen: “Physical Hypermedia: Organizing Collections of Mixed Physical and Digital Material”. Proceedings of the 14th. ACM International Conference of Hypertext and Hypermedia (Hypertext 2003), pp 10-19, ACM Press.

[25] K. Grønbæk, P.P. Vestergaard, P. Ørbæk: “Towards Geo-Spatial Hypermedia: Concepts and Prototype Implementation”. In proceedings of the 13th ACM Hypertext conference, June 11th - 15th, 2002, University of Maryland, USA, ACM, New York, 2002. pp 345 - 358.

[26] F.A. Hansen, N.O. Bouvin, B.G. Christensen, K. Grønbæk, T.B. Pedersen, J. Gagach: "Integrating the Web and the World: Contextual Trails on the Move". Proceedings of the 15th. ACM International Conference of Hypertext and Hypermedia (Hypertext 2004), ACM Press.

[27] S. Harper, C. Goble, S. Pettitt: "proXimity: Walking the Link". In Journal of Digital Information, Volume 5, Issue 1, Article No 236, 2004-04-07. Disponible en: http://jodi.ecs.soton.ac.uk/Articles/v05/i01/Harper/

[28] R.E. Horn: "Mapping Hypertex". Lexington Institute.USA.

[29] A. Knight, N. Dai: "Objects and the Web". IEEE Software, vol. 19, no. 2, pp. 51-59, March/April 2002.

[30] M.A. Kolb: "Developing Custom Tags for JSP". Caleo Networks, Austin, TX. www.wavedrag.com/blokware/space/Presentations/Css2000-CustomTags.ppt

[31] G.E. Krasner, S.T. Pope: "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System". ParcPlace Systems, Inc., Mountain View, USA, Tech. Rep., 1988.

[32] G. Landow, P. Delany: "Hypermedia and Literary Studies". Cambridge: Massachusetts Institute of Technology Press, 1991.

[33] W.E. Mackay: "Augmented Reality: Linking real and virtual worlds. A new paradigm for interacting with computers". Proceedings of the ACM Conference on Advanced Visual Interfaces (AVI 98), ACM Press, pp13-21.

[34] C. Marshall, F. Shipman: "Spatial hypertext: designing for change". In Communications of the ACM. 38, 8 (1995). pp. 88-97.

[35] C. Marshall, F. Shipman "Spatial hypertext and the practice of information". In "Proc. Tenth ACM Conference on Hypertext (Hypertext '97)". (Southampton, UK, Apr, 1997). pp. 124-133.

[36] C. R. McClanahan, T. Husted, Ed Burns: "Struts User and Developer Guide". http://struts.apache.org/userGuide

[37] C. McKnight, A. Dillon, J. Richardson: "Hypertext in Context". Cambridge University Press. 1991

[38] P. Milgram, F. Kishino: "A Taxonomy of Mixed Reality Visual Displays". IEICE Trans. Information Systems, vol. E77-D, no. 12, 1994, pp. 1321-1329.

[39] T. H. Nelson: "Literary Machines". Own publisher. 1990.

[40] C.O. Nueda: "Educación y futuro. Textos para una encrucijada". Colección Documentos de la Red. Consejería de Educación de la Comunidad de Madrid y Entimema. Madrid, 2001. ISBN 84-8198--367-5.

[41] X.A.I. Pello: "Struts. Implementación del patrón MVC en aplicaciones Web". JavaHispano. Marzo de 2002. http://www.abcdatos.com/tutoriales/tutorial/l5573.html

[42] W. Pree: "Design Patterns for Object Oriented Software Development". Addison-Wesley, 1994.

[43] L. Romero, N. Correia: "HyperReal: A Hypermedia model for Mixed Reality". Proceedings of the 14th ACM International Conference of Hypertext and Hypermedia (Hypertext 2003), pp 2-9, ACM Press.

[44] D. Schwabe, G. Rossi: “An object-oriented approach to web-based application design”. Theory and Practice of Object Systems (TAPOS), Special Issue on the Internet, v. 4#4, pp.207-225, October, 1998.

[45] Tutorial de Java - Arquitectura MVC
http://www.pablin.com.ar/computer/cursos/java1/Apendice/mvc.html

[46] The official Struts home page is at http://jakarta.apache.org/struts

[47] http://jakarta.apache.org/struts/api

[48] Disney Online - The Official Home Page of The Walt Disney Company: www.disney.com

[49] Walt Disney World Resort en la Florida: www.espanol.disney.com/disneyworld/index.html

[50] Disney Pix is a site dedicated to Disney: www.disneypix.com

[51] M3Gate WAP Browser: http://numeric.ru/distribbin/m3stp12.exe

[52] WAP Proof. WML browser for developers of mobile content: http://www.wap-proof.com/download/wproof_1_2_setup.exe