Principios y patrones de diseño de software en torno al patrón compuesto Modelo Vista Controlador para una arquitectura de aplicaciones interactivas.

Thesis · August 2010

DOI: 10.13140/RG.2.2.12554.26560

1 author:

Héctor Adrián Valdecantos
National University of Tucuman
5 PUBLICATIONS **7** CITATIONS

UNIVERSIDAD NACIONAL DE TUCUMÁN

Principios y patrones de diseño de software en torno al patrón compuesto Modelo Vista Controlador para una arquitectura de aplicaciones interactivas

por

Héctor Adrián Valdecantos

Director:

Ing. Alejandro Di Battista

Co-Director:

Ing. Ana Nieves Rodríguez

Una tesis realizada para completar el título de
grado de la carrera de Licenciatura en Informática

en la

Facultad de Ciencias Exactas y Tecnología

Departamento de Ciencias de Computación

23 de Agosto de 2010

UNIVERSIDAD NACIONAL DE TUCUMÁN

Abstract

Facultad de Ciencias Exactas y Tecnología
Departamento de Ciencias de Computación

Licenciatura en Informática

por Héctor Adrián Valdecantos

Estar preparado para entregar software de calidad en tiempo y forma de manera eficiente es una característica deseable en la industria. Se necesita de un marco de trabajo o framework que nos ayude a alcanzar esto en el desarrollo, es ahí donde aparece el patrón arquitectónico Modelo Vista Controlador para suplir las necesidades no funcionales de los sistemas interactivos. El trabajo duro es realizado sólo una vez al comienzo del proyecto, luego podremos concentrarnos en resolver los diferentes requerimientos funcionales que el sistema debe alcanzar.

Aplicando el patrón compuesto MVC podremos dividir el equipo de desarrollo de software para que se especialice en distintas partes por separado, será posible obtener un grado aceptable de reutilización, que favorecerá a la extensibilidad, portabilidad, y el mantenimiento del software.

Entender un concepto de alto nivel de abstracción como el patrón arquitectónico MVC para crear un framework es solamente posible a través del entendimiento de los patrones de un nivel de abstracción más bajo que lo componen. Todos los patrones están basados en el uso de los principios que rigen la construcción de software, los cuales debemos conocer porque son fundamentales para aplicar estos patrones exitosamente.

Primero se expondrán los problemas que pueden existir al no tratar de manera debida las cuestiones relacionadas con la interactividad en los sistemas. Luego, en forma progresiva, se explicarán conceptos básicos de arquitectura, principios, y patrones de software, en torno a MVC para encarar el patrón compuesto que describe esta tesis.

Agradecimientos

Agradezco a Alejandro por haber aceptado dirigir esta tesis y destinar parte de su tiempo como profesional a la supervisión de este trabajo. Además, por transmitir su pasión y estar al tanto de las tendencias en la tecnología de información.

A Ana que co-dirigió y siguió con mucha atención e interés el desarrollo de la tesis, y supo siempre responder inmediatamente.

Al departamento de Ciencias de Computación, y a todos aquellos que en algún momento supieron escuchar o dar una opinión acerca de mis comentarios sobre los temas en que estuve trabajando para escribir la tesis...

Índice

Índice de figuras

Índice de cuadros

Abreviaciones

API	**A**pplication **P**rogramming **I**nterface
GUI	**G**raphic **U**ser **I**nterface
HCI	**H**uman **C**omputer **I**nteraction
HTML	**H**yper**T**ext **M**arkup **L**anguage
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
MVC	**M**odel **V**iew **C**ontroller
QA	**Q**uality **A**ssurance
SQL	**S**tructure **Q**uery **L**anguage
UML	**U**nified **M**odeling **L**anguage

Introducción

Esta tesis no aporta elementos nuevos para la comunidad, su verdadero aporte es esclarecer conceptos y guiar al lector a un entendimiento progresivo del patrón central que trata. Hoy en día el patrón Modelo Vista Controlador o MVC forma parte del vocabulario común de los desarrolladores, y si no es así, debería.

Cualquier persona que no ha incursionado en los principios, ni en los patrones de software, pero tenga los conceptos en claros del paradigma orientado a objetos, además disponga de alguna experiencia en el desarrollo de pequeñas aplicaciones con algún lenguaje de programación que soporte de alguna manera este paradigma, y pueda interpretar básicamente diagramas UML[1] de clases y secuencia, está en condiciones de leer esta tesis y entender sin problema los temas que trata.

La comunicación es algo muy importante en el desarrollo de software, porque es imposible crear software si no lo hacemos en grupo con otras personas. Cuando nos comunicamos siempre manejamos un grado de abstracción, que necesariamente debe coincidir con el grado de abstracción que maneja la persona que nos escucha, sino entraríamos consecuentemente en conflicto con lo que se trata de comunicar. Mucho más importante es manejar correctamente los conceptos que las palabras hacen referencia. Desafortunadamente muchas veces se toma un vocabulario que no se conoce adecuadamente y esto crea indefectiblemente un problema en la comunicación. Se usan palabras que parecen impactantes y creemos le dan fuerza o importancia a un tema o trabajo, pero tal vez, no se es capaz de abarcar y capturar su esencia. Por todo esto pondré especial esfuerzo en desarrollar conceptos y usar el vocabulario como corresponde, y cuando parezca necesario siempre habrá una referencia para seguir leyendo sobre el tema.

Como esta tesis trata sobre patrones, y se quiere ser progresivo en la exposición de los conceptos, se incluyó un espacio para tratar un tema muy relacionado con los patrones, como el de los principios de diseño. En la vida misma, tener en claro los principios que seguimos y no solo el objetivo final al que queremos llegar facilita que la transición desde el punto de partida hasta este objetivo se desarrolle de un modo correcto y nos permita

[1] "Unified Modeling Language", o Lenguaje de Modelado Unificado (http://www.uml.org/).

extender, ampliar o alcanzar objetivos mayores. Por supuesto, en esta tesis se habla de tener en claro los principios del diseño orientado a objetos para obtener diseños flexibles, mantenibles y reutilizables.

Los problemas de implementación de las interfaces de usuario atraviesan todos los niveles de abstracción que existen en un sistema. Si estos problemas no son solucionados de manera correcta, seguro llegará a empobrecer el sistema y a sus desarrolladores. No quedarán dudas de la necesidad de una solución arquitectónica después de considerar todos los problemas en que nos veríamos envuelto si no tratamos adecuadamente el problema de las interfaces de usuario.

Como recalcan muchos escritos muy importantes en el área de diseño de software orientado a objetos, los cuales muchos fueron fuente para esta tesis, aquí no haré la excepción, y diré que desarrollar software es una tarea difícil, y realizar software reutilizable más difícil aún. Se necesita experiencia, pero no solo se adquiere experiencia creando software, sino también exponiendo nuestros conocimientos.

Como dije, no puedo declarar que esta tesis haga un aporte novedoso a la comunidad. Todos los principios que voy a desarrollar ya han sido publicados, y los patrones que expondré ya han sido clasificados y catalogados. Espero que la lectura de esta tesis ayude a construir conceptos y un vocabulario común en cuanto a principios y patrones se refiere, y que al mismo tiempo ayude a la comunicación y a la enseñanza.

Estructura de la tesis

La tesis está estructurada de manera que el desarrollo de los conceptos sea progresivo, por lo tanto se aconseja una lectura respetando el orden de los capítulos, o comenzando desde el capítulo que no se disponga un entendimiento de los temas que trata. En los primeros cuatro capítulos se propone lograr un acercamiento teórico al tema, mientras que los capítulos 5 y 6 son esencialmente prácticos donde se muestran ejemplos de aplicaciones de principios y patrones respectivamente.

- Capítulo 1
  Presenta el problema de la implementación de las interfaces gráficas de usuarios y limita el dominio del tema que trata la tesis.
- Capítulo 2
  Enuncia la solución que se desarrollará a lo largo de la tesis, exponiendo brevemente, a priori, conceptos fundamentales de arquitectura de software, su valor y necesidad de definirla.

- Capítulo 3
  Enumera algunos importantes principios del diseño de software y se aclaran conceptos con algunos pequeños ejemplos, sin antes haber definido lo que es un principio.
- Capítulo 4
  Explica cuestiones relacionadas con los patrones de software en general, sus características, descripción, y clasificación.
- Capítulo 5
  Visita varios patrones, los implementa, y desarrolla ejemplos tangibles, exponiendo los principales principios relacionados con cada patrón.
- Capítulo 6
  Muestra el patrón central de la tesis en su forma clásica, y detalla paso a paso su aplicación con un ejemplo sencillo para ver todos los mecanismos involucrados.
- Capítulo 7
  Concluye sobre el tema, las ventajas de la aplicación del patrón MVC y el uso de un framework, y resalta algunas de sus características principales.

Ejemplos

Se eligió el lenguaje de programación JAVA para implementar los ejemplos de esta tesis por estar relacionado de forma cercana con el lenguaje C++ que es de uso común en el ámbito de la universidad origen[2], además por presentar una sintaxis casi idéntica y librar al programador de complejidades técnicas que de otro modo debería resolver.

Software Libre

Para la realización de esta tesis se hizo uso, únicamente, de software libre bajo una plataforma Linux. Entre otros, se usó:

Latex : http://www.latex-project.org/
Texworks: http://www.tug.org/texworks/
Bouml: http://bouml.free.fr/
Dia: http://projects.gnome.org/dia/
Gimp: http://www.gimp.org/
NetBeans: http://netbeans.org/
Java SE: http://java.sun.com/javase/

[2] Al día de la fecha, C++ es usado como lenguaje de programación en varias materias de la carrera de Licenciatura en Informática de la Universidad Nacional de Tucumán, Argentina.

Capítulo 1

Interfaz de Usuario

> *"It is the system to most users. It can be seen, it can be heard, and it can be touched. The piles of software code are invisible, hidden behind screens, keyboards, and the mouse. The goals of interface design are simple: to make working with a computer easy, productive, and enjoyable."*
>
> Wilbert O. Galitz

Como interfaz de usuario entendemos la parte visible de un sistema de información, de la cual el usuario mismo puede armar un modelo mental del sistema que esta utilizando.

La interacción hombre-computador o, más conocido como HCI (Human-Computer Interaction) por su sigla en inglés, es una disciplina concerniente al diseño, evaluación, e implementación de sistemas computarizados interactivos para el uso humano y el estudio de los principales fenómenos alrededor de ellos. [HBC+96]

Cuando hablamos de interfaz de usuario, hablamos de las partes encargadas de tratar todas las cuestiones de la interactividad con el sistema.

1.1. La creciente complejidad del software

Todo software útil tiende a evolucionar en el tiempo, a adaptarse a su cambiante contexto para satisfacer las crecientes demandas que el mundo real impone. Especialmente con el constante crecimiento de Internet, donde la aparición de la web 2.0 incrementó la demanda de aplicaciones ricas en la web, las cuales exigen una compleja interacción con el usuario.

Si bien el desarrollo de software nunca se trató de una tarea solitaria, hoy en día lo es menos aún debido al alto grado de complejidad que puede alcanzar el software actual. Se requiere de un grupo de trabajo, donde además de aplicar una ingeniería del software que permita la sinergia del equipo de desarrollo para orientar sus fuerzas, el producto mismo que se esta desarrollando debe permitir esta ingeniería. Como ingeniería del software entendemos no solo a la definición de los procesos y métodos que guían el desarrollo, sino también las distintas técnicas que permiten una ingeniería, como el versionado o source control, testing unitario, testing de integración, integración continua, continuous build, bugtracking, issue tracking for QA y sistemas colaborativos para la comunicación. No se puede hacer nada en potenciar el desarrollo del software con una ingeniería si el producto producido no permite ninguna ingeniería en absoluto.

Los diversos documentos o productos de software que generemos van a depender de la metodología usada para su desarrollo. Estos documentos o productos van a variar en forma, en cantidad, y en existencia. Podemos decir que la documentación que se generará podrá ser, entre otros, algunos documentos de requerimientos, diferentes diagramas de análisis y diseño, definición de procesos, pero principalmente se generará, sin depender de la metodología usada, código fuente como producto de software.

La ingeniería de software es necesaria para atacar la complejidad del desarrollo de sistemas, pero si el código fuente que generamos no permite que el sistema sea desarrollado en forma separada: ¿de qué nos sirve un equipo? ¿De que nos sirve una ingeniería? Si el código nos dificulta, o no nos permite la integración con una solución ya construida, o sea reutilizar software, ¿como potenciamos y evolucionamos en el desarrollo?

1.2. El problema de las Interfaces Gráficas de Usuario

Unos de los problemas que plantean las interfaces gráficas de usuario o GUIs (Graphic User Interface) de los sistemas es el diseño de la interacción efectiva para lograr una buena experiencia de usuario. Un buen diseño de interfaces de usuarios requiere conocer características humanas: cómo vemos, cómo entendemos y cómo pensamos. Requiere

definir cómo la información debe ser presentada, generalmente en forma visual, para mejorar la aceptación y comprensión humana. Estos problemas, que no son menores, están relacionados directamente con la experiencia que el usuario obtiene al usar la interfaz, es la interfaz desde la perspectiva del usuario en cuanto a su usabilidad. Estos tipos de problemas no serán tratados en esta tesis.

El problema que se trata de presentar aquí es en relación con la implementación de las interfaces de usuarios. Es el problema relacionado con la pila de código fuente de software que existe atrás de una interfaz, del cual un sistema de información se vale para lograr la interacción entre el usuario y el sistema. La interfaz desde la perspectiva del programador, del desarrollador, y hasta del arquitecto de software es lo que se tratará de aquí en adelante.

Desde esta perspectiva, al momento de implementar interfaces de usuario, hay que preguntarse: ¿En qué grado el problema de las interfaces afecta al sistema? ¿Es un problema que afecta solamente a los desarrolladores de las interfaces, al sistema, o es un problema que también puede afectar al proceso de desarrollo?

1.2.1. En el desarrollo de software

No importa cuán definido tengamos un proceso de desarrollo de software, lo que realmente refleja y logra la reificación[1] del mismo es, entre otras cosas, el producto final y principal que generamos, que representa el software que creamos, o sea el código fuente. Este producto revela de alguna manera este proceso y lo acompaña.

En lo que respecta a las GUIs, sería caótico trabajar en equipo y producir un código fuente donde en un único archivo convivan aspectos de la presentación, el procesamiento, y manejo de datos. Podríamos presentar un formulario al usuario, capturar las entradas del usuario, conectarnos a una base de datos, buscar información relevante en base a las entradas del usuario, hacer el proceso necesario que corresponda según la lógica de negocio, y devolver los resultados. Esto es posible, pero para nada recomendable ya que entorpecería el desarrollo de software en paralelo de los distintos integrantes del equipo.

Se necesita disciplina, un proceso, y aprender a desarrollar para poder trabajar en paralelo con otros en diferentes aspectos de un mismo proyecto para que al momento de la integración el sistema final sea un todo continuo[2].[Abe09]

[1] Convertir algo en cosa, puede entenderse como un proceso de transformación de representaciones mentales en algo concreto. Se puede decir que un objeto es la reificación de una clase.

[2] Del término en inglés 'seamlessly' que hace referencia a un todo sin costura, de una sola pieza donde no se noten las juntas de entre las diferentes partes que lo componen.

Para asegurar la calidad del producto de software algunos desarrolladores incluyen la creación de tests automatizados como parte de su desarrollo, ya sea para las partes criticas, o por qué no, para la totalidad del sistema. En el contexto de la interacción con el usuario resulta una ventaja, al momento de crear los tests, el tener bien definido el mecanismo que el sistema va a usar para presentar las GUIs, para capturar las acciones o entradas del usuario, y para procesar la información.

En conclusión, para desarrollar software no solo necesitamos una metodología, sino un plan que establezca las estructuras principales a partir de la cual se desarrollará y organizará el software.

FIGURA 1.1: Mapa piloto a partir del cual se desarrollaría Brasilia.

1.2.2. En la reutilización de software

La reutilización del código va en contraposición del código duplicado, y es la única forma de alcanzar una producción y calidad adecuada en la industria del software actual.

Un diseño es inmovible o inmoble cuando contiene partes que podrían ser útiles en otro sistema, pero el esfuerzo y el riesgo de separar estas partes del sistema original son muy grandes. Esto es muy desafortunado pero muy común.[RM06]

Muchas veces, cuando se trabaja utilizando herramientas para el desarrollo rápido de interfaces del tipo WIMP (Windows, Icon, Mouse, and Pointer) como Visual FoxPro, Visual Basic, Delphi, Powerbuilder, y otras, por lo general creamos código que trata

diferentes aspectos del sistema de una manera muy acoplada. Estas herramientas fueron diseñadas para desarrollar sistemas con interfaces de usuarios que permitan, entre otras cosas, manejar fácilmente el acceso y escritura a bases de datos relacional. Nos brindan objetos o componentes, como *ComboBox*, tablas, grillas, listas, áreas de texto, que pueden ser arrastrados a un área de diseño y así componer el layout[3] de una interfaz gráfica de usuario, para luego ser ligados o conectados a una base de datos por medio de propiedades provistas de manera visual, donde podemos especificar el nombre de la base de datos, el tipo de conexión, la consulta SQL[4] que debe ejecutar, y demás propiedades.

Si bien este tipo de herramientas tuvo mucho éxito, y aún es usada considerablemente, solo nos permite crear sin dificultades aplicaciones que solamente actualicen y visualicen datos de una base de datos relacional. Trabajar de esta manera crea una fuerte dependencia entre dos aspectos totalmente diferentes como la presentación y el acceso a datos. Una vez que nuestra aplicación requiere de una lógica de negocio mas complicada, con validaciones y cálculos complejos, usualmente los usuarios de estas herramientas embeben esta lógica directamente en el código de las interfaces de usuario. Luego, ante nuevos requerimientos, y la imposibilidad de separar los aspectos mencionados por haber trabajado de ésta forma, normalmente se termina realizando una duplicación de código para resolver partes de problemas que tal vez ya se habían resuelto para otra interfaz.

FIGURA 1.2: Marcel Duchamp toilet ready-made, dada movement, 1917.

Este problema se hizo muy visible con el auge de las aplicaciones web, donde empresas con sistemas que tenían su lógica de negocio pegada o acopladas a las interfaces de usuarios se vieron en la imposibilidad de usar esa lógica ya construida en la web. Esta

[3] El layout de una interfaz hace referencia a la composición de los elementos visuales de una pantalla.
[4] Structured Query Language o Lenguaje de Consulta Estructurado.

imposibilidad estaba dada por el cambio en la tecnología de presentación usada hasta ese momento, se debía pasar de un sistema de ventanas, menús, iconos, y gráficos que dependía de recursos nativos de la plataforma del sistema operativo, a una tecnología web donde las interfaces de usuarios son creadas de una forma totalmente diferente, usando el lenguaje HTML bajo el protocolo HTTP para lograr un mecanismo de presentación independiente de la plataforma.

1.2.3. En la extensión del software

Un sistema que sea extensible para asegurar la provisión confiable de nuevas funcionalidades deseadas debe poseer un orden y características que permitan la reutilización de las partes que lo componen. El modo en que implementamos las GUIs puede entorpecer la extensión de sistemas a nuevas funcionalidades por las mismas razones que pueden entorpecer la reutilización del software.

La extensibilidad es una medida sistémica de la habilidad de extender un sistema y el nivel de esfuerzo requerido en implementar la extensión. Un sistema se puede extender a través de la agregación de nuevas funcionalidades, o la modificación de funcionalidades existentes.

La rigidez es la tendencia del software a ser difícil de modificarlo, aún en formas simples. Un diseño es rígido si un solo cambio causa una cascada de cambios subsecuentes en módulos dependientes. Cuando más módulos tienen que ser cambiados ante una modificación, mas rígido es el diseño.[RM06]

FIGURA 1.3: Juego de piezas plásticas, Rasti.

En los sistemas, la extensibilidad quiere decir que el diseño incluirá un grado de genericidad o generalidad adecuado que permita su extensión sin tener que realizar cambios drásticos. Es un plan para la porción del camino aún no recorrido, pero que es muy probable de transitar en un futuro.

Lograr un sistema extensible y reutilizable es difícil, y requiere un esfuerzo mayor lograr mecanismos y elementos genéricos en el software que tal vez nunca sean utilizadas. Por esto hay que lograr un grado de genericidad adecuado y balanceado según las expectativas del software que se está desarrollando.

La extensión amplía el área funcional de un sistema, mientras que la expansión sólo logra completitud y correctitud en las funciones existentes sin agregar nuevas funcionalidades.

Los mecanismos de implementación de interfaces de usuarios en un sistema deben poseer un grado de genericidad para facilitar la extensión del sistema a nuevas funcionalidades para los usuarios, y a la vez evitar la rigidez en el software.

1.2.4. En la portabilidad del software

La capacidad de mover un sistema de software a diferentes plataformas con una mínima inversión define su portabilidad. Un sistema portable va a necesitar muy pocos cambios para que esté disponible en un entorno distinto.

FIGURA 1.4: Charles Spencer Chaplin con su bastón y sombrero bombín.

El código fuente que representa los elementos de las interfaces gráficas como ventanas, iconos, cuadros de dialogo, entre otros, están fuertemente acoplados a recursos gráficos brindados por los distintos sistemas operativos y hardware. Si tenemos embebida la lógica de negocio en el código de las interfaces, será impensable portar nuestro sistema a otra plataforma operativa que maneje recursos gráfico no compatibles, o a una tecnología de presentación totalmente diferente.

1.2.5. En el mantenimiento del software

En la ingeniería del software, la mantenibilidad es una característica marcada por la facilidad que un producto de software puede ser modificado, ya sea para corregir defectos o cumplir con nuevos requerimientos.

La fragilidad es la tendencia de un programa de romperse o quebrarse en varios lugares cuando se realiza un cambio. A veces, los problemas están en áreas que no tienen relación conceptual con el área donde se realizó el cambio. Arreglar estos problemas lleva aún a más problemas... [RM06]

En un sistema de información, el tiempo que nos toma encontrar e identificar un bug[5] es una medida de cuán mantenible es el sistema, más que de cuánto tiempo nos lleva solucionarlo. También es una medida de mantenibilidad el grado en que la solución a un bug afecta al sistema, o produce una regresión en el desarrollo.

FIGURA 1.5: Gotas de agua sobre la hoja ultrahidrofóbica de la planta taco de reina.

El código fuente de las interfaces de usuarios mezclado con la lógica de negocio, y acceso a base de datos, es un síntoma de lo costoso y difícil que resultará el mantenimiento

[5] "Computer bug" para indicar un defecto de software que es el resultado de un fallo o deficiencia durante el proceso de desarrollo.

del sistema, ya que las interfaces de usuario son siempre propensas a cambios. Desde cuestiones estéticas a funcionales afectarán al código fuente de las interfaces de usuario. Por ejemplo, cuando extendamos la funcionalidad de una aplicación, los menús tendrán que ser modificados para acceder a las nuevas funciones. Las interfaces de usuarios también podrían tener que ser adaptadas para usuarios específicos. Un sistema, por diferentes razones, podría ser portado a una nueva plataforma con un diferente *look and feel*[6]. El solo hecho de cambiar el sistema de ventanas de la plataforma requeriría un cambio en las interfaces.

El código que corresponde a las interfaces de usuarios de cualquier sistema, está en una posición de constantes cambios, y lograr que este código sea mantenible resulta en un gran beneficio para el futuro.

1.2.6. En la escalabilidad del software

Logramos escalabilidad en el software cuando el diseño de un sistema funciona para diferentes escalas. Si una aplicación que ponemos en marcha es accedida por cientos de usuarios y luego de un tiempo es accedida por cientos de miles de usuarios con la misma eficiencia y con mínimos cambios, se puede decir que esa aplicación es escalable. Ahora bien, el software por sí mismo no logra escalabilidad, lo hace gracias a mayores recursos que pueda disponer. Entonces, escalabilidad es una medida de cómo la adición de recursos, usualmente de hardware, afecta la performance o rendimiento de un sistema.

Un sistema escalable permite agregar hardware, como agregar memoria, procesadores, o aumentar la cantidad de servidores disponibles, y de esta forma obtener un mejor rendimiento en el sistema. Hacemos un scaling-up (escalada vertical) cuando agregamos más recursos a un mismo servidor, y scaling-out (escalada horizontal) cuando agregamos más servidores o nodos.

¿Pero cómo un software puede ser escalable? Gran parte de la escalabilidad de un sistema es resuelta por la plataforma donde se encuentra implantado o desplegado[7], pero también puede ser resuelta y prevista en su diseño. Debe ser resuelta en varios niveles de abstracción del sistema, desde la programación, cuando programamos en forma concurrente con diferentes hilos y procesos, hasta en un nivel arquitectónico, diseñando para poder separar distintas partes y destinarlas a ser ejecutadas en diferentes servidores.

[6] En el diseño de software "look and feel" es usado en el ámbito de una interfaz gráfica. El 'look' o apariencia comprende aspectos de su diseño estético, inclusive elementos como colores, formas, disposición, tipo de letras, mientras que el 'feel' o comportamiento hace referencia a los aspectos dinámicos como las acciones sobre botones, menús y demás elementos de una interfaz.

[7] Se suele usar el verbo 'deploy' en inglés para hacer referencia a la acción de implantar un sistema.

Cuando estamos ante interfaces de usuarios muy acopladas con la lógica de negocio, y con el manejo de la persistencia de los datos, se hace muy difícil lograr que el sistema completo funcione eficientemente en escalas mayores. La única forma posible de escalar un sistema de éstas características es realizando una escalada vertical en el hardware.

Si bien las GUIs no afectan directamente la escalabilidad de un sistema, una buena implementación de éstas facilitaría incorporar los mecanismos necesarios para escalar una aplicación de manera más sencilla.

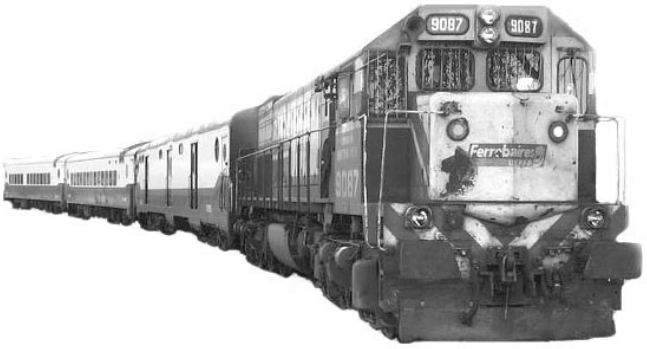

FIGURA 1.6: Tren de Ferrobaires, transporte de pasajeros interurbano. Gobierno de la provincia de Buenos Aires.

1.3. Conclusión

Si queremos encontrar una solución de software en el paradigma orientado a objetos a problemas complejos, que permita un desarrollo viable en paralelo, que sea fácilmente mantenible ante fallas o nuevos requerimientos sin ocasionar regresiones mayores en su elaboración, que sea factible reutilizar algunas de las partes ya construidas para continuar con la construcción de la aplicación, que permita su extensión a nuevas funcionalidades, que sea posible su portabilidad a diferentes plataformas evitando el rediseño o minimizándolo, y que acepte un cambio de escala ante un aumento en la carga de peticiones que el sistema deba soportar, debemos lograr un diseño flexible y reutilizable para nuestra aplicación. Es aquí donde radica la verdadera complejidad del software, en alcanzar todas estas características deseables.

Capítulo 2

Solución Arquitectónica

"L'architecte, c'est formuler les problèmes avec clarté.[1] *"*

Le Corbusier

Cuando vemos que un problema, como la implementación de las GUIs, afecta de múltiples formas el desarrollo de un sistema, desde el trabajo en equipo, la reutilización, la extensibilidad, la portabilidad, la mantenibilidad, y hasta indirectamente la escalabilidad del software, estamos ante un problema que su solución debe dirigir y organizar todas las partes del sistema que estén relacionadas con el aspecto que se trata solucionar. Este tipo de soluciones, son soluciones de alto nivel de abstracción, soluciones que abarcan, tocan, o atraviesan transversalmente[2] el sistema que pretendemos crear. En definitiva, estamos ante búsquedas de soluciones de carácter arquitectónico, las cuales establecen y pautan un orden para el resto de los elementos que conforman sistema.

[1] El arquitecto, es el que formula los problemas con claridad.

[2] "Cross-cutting issues" es el término en inglés usado para nombrar cuestiones que atraviesan transversalmente diferentes partes del sistema.

2.1. Arquitectura de Software

La arquitectura del software está vista como una importante subdisciplina de la ingeniería del software, particularmente en el campo del desarrollo de grandes sistemas de información. [CBB+02]

Mucho autores sospechan o siembran dudas con respecto al entendimiento del concepto de arquitectura en el área del software, plantean que hay una variada interpretación de este término que hasta a veces resulta contradictoria. También coinciden en que es una palabra sobre usada, que suena a algo imponente, y que indica que se está hablando de algo importante.

Tal vez no se encuentra un consenso último del término arquitectura en el área del software debido a su amplio uso a través de siglos en el área de la construcción edilicia. Tal vez el término se acerque más ahora a su origen con el auge del área de las tecnologías de información, donde su uso es ya muy requerido, y se despegue, de esta manera, del área de la construcción de edificaciones o espacios urbanos.

Definitivamente la arquitectura es algo importante, que está relacionada con la construcción inteligente y planeada, para resolver un problema sin pensar en detalles. Es la descomposición de alto nivel de abstracción de un sistema en sus partes, la especificación de las relaciones entres las partes, y todas las decisiones de construcción que serán difíciles de cambiar después de construido el sistema.

Si bien el software está hecho de un material blando, plástico y maleable, hasta intangible podríamos decir, hablamos de arquitectura para definir las parte del software que son diseñadas con inteligencia, y se estiman perdurarán durante gran parte de la vida del sistema de software. Aquí es donde tal vez encontramos una analogía con lo que comúnmente se entiende por arquitectura en el ámbito de la construcción edilicia, diciendo que nos referimos como arquitectura a la parte dura o que no cambia del software. Cuando encontramos una analogía que refuerza en alguna parte el concepto del término que estamos usando, casi automáticamente lo aceptamos con más tranquilidad, pero tal vez provocamos un desvió del concepto que estamos construyendo.

Entonces, la arquitectura es la parte del software que no cambia con frecuencia, y que de algún modo organiza los demás elementos constructivos del sistema.

The architecture establishes constraints on downstream activities.[CBB+02]

Realmente parece no haber una definición de arquitectura del software con una amplia aceptación en la industria, es por eso que un buen lugar para ver varios puntos de vistas

sobre su definición es en el sitio web del Software Engineering Institute, del Carnegie Mellon:

www.sei.cmu.edu/architecture/start/community.cfm

2.1.1. Etimología

El término arquitecto proviene del antiguo idioma griego arqui-tectón «primero-obra», que significa literalmente el primero de la obra, o máximo responsable de una obra. La palabra "Arquitecto" de origen Griego suele tener distintas interpretaciones dependiendo de la bibliografía a la cual se recurra. En la Enciclopedia Encarta dice que viene por Jefe y Teckto de Carpintero u obrero. En libros de origen inglés, «Arq» es un superlativo, como en el caso de Arz-obispo, más que un obispo, o Archi-criminal, más que un criminal, Archi-teckto sería «más que un constructor». Hay que tener en cuenta que la palabra y profesión de Arquitecto es milenaria y el actual Título de Arquitecto tiene menos de tres siglos. [WIK]

2.2. Arquitectura y diseño

La arquitectura es diseño, pero no todo diseño es arquitectura. Muchas decisiones de diseño en un sistema de información no son parte de la arquitectura, estas decisiones son delegadas a diseñadores, implementadores o programadores, que resuelven problemas más técnicos con un nivel más bajo de abstracción.

Si una estructura es necesaria para suplir las necesidades de un sistema, entonces esa estructura es arquitectura. Al diseñar un subsistema, las personas asignadas a esa tarea pueden crear nuevas estructuras para alcanzar sus propios objetivos. Estas estructuras, según su visión, se corresponden a una arquitectura, pero no forma parte de la arquitectura de visión más general.

El arquitecto define el limite entre lo que es y no es un diseño arquitectónico. Define una arquitectura para que el proyecto de sistema pueda cumplir con sus necesidades de desarrollo, de comportamiento, y de calidad.

2.3. Definición estructural

El arquitecto de software pasa gran parte de su tiempo pensando en cómo partir su aplicación en partes interrelacionadas entre sí, ya sea en capas, módulos, componentes,

u objetos. Cualquiera fuese la unidad de partición, también debe definir la forma en que estas partes se interrelacionan.

Cuando se realiza la partición, el arquitecto asigna responsabilidades a cada parte. Estas responsabilidades definen las tareas que cada partición debe realizar dentro de la aplicación, y a su vez especifica qué rol cumple cada parte en el sistema. Las partes deben poder ser construidas casi independientemente de las otras partes, aunque al final deben ser reunidas para resolver el problema mayor. Un solo sistema puede estar particionado simultáneamente en un numero diferentes de formas para resolver diferentes aspectos, o sea, un sistema puede tener múltiples arquitecturas que definen diferentes estructuras para resolver diferentes aspectos de una misma aplicación.

Una cuestión estructural clave para casi todas las aplicaciones es minimizar las dependencias entre las partes, creando una arquitectura desacoplada o débilmente acopladaa partir de partes o componentes altamente cohesivos. Eliminando dependencias innecesarias, cualquier cambio que se realice será localizado y no se propagará a través de todo el sistema.

Hay que evitar las excesivas dependencias ya que éstas dificultan realizar cambios en el sistema, haciendo más costoso realizar los test, más largo el tiempo de *build*[3], y hace que el desarrollo concurrente sea más duro y difícil de realizar.

2.4. Comunicación

Cuando una aplicación es dividida en un conjunto de componentes, se vuelve necesario pensar cómo esos componentes se van a comunicar y relacionar. La arquitectura debe especificar cómo sus partes se comunican, y cómo la información va a fluir entre estas partes. La arquitectura es lo que hace que un conjunto de partes trabajen juntas como un todo exitoso.

Las partes o componentes de una aplicación pueden ejecutarse en un mismo espacio de nombres, donde todos sus componentes son locales, o también pueden ejecutarse en diferentes espacios de nombres de manera remota. También hay que tener en cuenta que su ejecución puede ser en diferentes hilos o procesadores, y la comunicación debe realizarse mediante mecanismos de sincronización.

[3]Término usado para nombrar el proceso de convertir código fuente a código de máquina, similar a una compilación pero más abarcativo, ya que la construcción o build se aplica a múltiples archivos en uno o más proyectos e incluye el proceso de linkeo.

2.5. Requerimientos no funcionales

Los requerimientos no funcionales son aquellos que, más que definir qué hace una aplicación, están involucrados en cómo la aplicación provee los requisitos funcionales. La arquitectura de software está orientada a resolver requerimientos no funcionales.

Hay tres áreas generales distintas en los requerimientos no funcionales de un sistema de software: las que plantean restricciones técnicas, las que tratan las restricciones de negocio, y las que están relacionadas con la calidad.

- Los requerimientos no funcionales técnicos están restringidos puramente por cuestiones técnicas, como lo son las especificaciones tecnológicas. Por ejemplo el uso de un determinado lenguaje, la disposición de un framework de desarrollo, o hasta la tecnología que cuenta un determinado servidor. Estos requerimientos no son negociables, es lo que se dispone.
- Los requerimientos no funcionales relacionados con cuestiones empresariales económicas también restringen las decisiones de diseño de un arquitecto, pero no por razones técnicas. Por ejemplo, en el caso de un sistema distribuido, donde la empresa que provee un *middleware* eleva sus precios y nos vemos obligados a cambiar a un middleware opensource.
- Los atributos de calidad definen los requerimientos de una aplicación en términos de escalabilidad, disponibilidad, mantenibilidad, portabilidad, usabilidad, performance, entre otros. Estos requerimientos están relacionados no solo con los usuarios del sistema, sino también con el equipo de desarrollo.

La arquitectura de una aplicación debe cumplir con estos requisitos que van a influir en las decisiones de diseño. Aun así, el arquitecto necesita entender los requerimientos funcionales que el sistema debe cumplir, para definir una arquitectura que organice la construcción de los elementos encargados de suplir estos requerimientos funcionales, y que simultáneamente satisfaga los requerimientos no funcionales.

2.6. Arquitectura como abstracción

La arquitectura establece restricciones en actividades de más bajo nivel, y estas actividades pueden producir artefactos que se adecuan o responden a la arquitectura, pero la arquitectura no define una implementación. La arquitectura es puramente un elemento

del dominio de la solución, debe su propia existencia a la solución de los problemas de índole arquitectónicos.

Cuando se describe una arquitectura, es necesario emplear abstracciones para que pueda ser entendible por el equipo de desarrollo y los interesados en el proyecto. Esto se hace típicamente describiendo cada parte del sistema como una caja negra, especificando solamente sus propiedades desde afuera.

Una técnica muy usada para describir una arquitectura es la descomposición jerárquica denotada por los diferentes niveles de abstracción. Las partes que aparecen en un nivel de descripción son descompuestas luego más detalladamente. Estos diferentes niveles de descripción son necesarios para diferentes personas dentro del equipo. La arquitectura realiza una partición clara de las responsabilidades de las partes de un sistema para que puedan ser desarrolladas por diferentes equipos.

2.7. Una solución al problema de las GUIs

El aspecto de nuestro problema en el sistema es el que está relacionado con la interacción entre hombre y máquina, el de desarrollar software con una interfaz para usuarios humanos, el de crear aplicaciones interactivas.

Señalamos, en el Capítulo 1, todas las situaciones posibles que pueden surgir, o por las que podemos pasar, en consecuencia de un diseño de software pobre, escaso o nulo en lo que respecta a la implementación de las interfaces de usuarios. Vimos, que estos problemas no son mínimos y repercuten negativamente en el desarrollo y mantenimiento de todo el sistema como para dejarlos de lado.

Como características deseables en un sistema, queremos:

- Una solución de software en la que las interfaces de usuarios sean fácilmente modificables, y hasta intercambiables en tiempo de ejecución.
- Queremos que diferentes partes del equipo de desarrollo pueda trabajar en forma paralela, concurrente e independiente en un mismo proyecto. Una parte del equipo puede dedicarse a la lógica de negocio, donde pueden llegar a ser expertos en el dominio del problema sin preocuparse por la presentación o cara que el sistema mostrará a los usuarios. Otra parte del equipo debería poder tratar todos los problemas concernientes a la experiencia del usuario a través de las interfaces gráficas, los cuales podrían trabajar junto a diseñadores gráficos, artista, y expertos en temas relacionados con la interfaz desde la perspectiva del usuario, sin preocuparse por cómo el sistema debe procesar la interacción del usuario.

- Adaptar o portar la interfaz de usuario a otra plataforma no debería impactar en el código correspondiente a la lógica de negocio de la aplicación. Debemos poder utilizar diferentes tecnologías de presentación para las interfaces en diferentes plataformas sin necesidad alguna de modificar los objetos del dominio que modelan nuestra aplicación.
- Hacer correr diferentes partes del sistemas en diferentes servidores para favorecer la escalabilidad horizontal o *scaling out*, para que el sistema pueda seguir funcionando en escalas mayores.

En definitiva, estamos diciendo que el problema de las interfaces de usuarios requiere una solución de nivel arquitectónico.

La solución básicamente consiste en adherirse al principio de diseño de separación de responsabilidades. Necesitamos una estructura que separe las responsabilidades de presentación de las interfaces de usuario, las responsabilidades de control de la interacción del sistema con el usuario, y las responsabilidades del modelo que hará toda la computación necesaria para devolver resultados. La solución también debe especificar en qué modo estas partes se comunicarán e interrelacionarán entre sí.

La interfaz presenta la aplicación al usuario, y basado en las entradas que el usuario produce, se ejecutan distintos procesos relacionados con las reglas de negocio del sistema.

Una vez que el problema se descompone en base a la separación de responsabilidades, podemos poner a trabajar a diferentes miembros del equipo en diferentes aspectos de una misma aplicación.

Para lograr estas características en la solución, vamos a aplicar el patrón arquitectónico de diseño de software conocido como Modelo-Vista-Controlador[4]. Este patrón hace uso del principio de separación de responsabilidades, logrando una separación de partes que atacarán los problemas de la implementación de las interfaces de usuarios. Define cómo estas partes se relacionan, y crea una estructura que organiza los demás elementos que constituyen el sistema.

Estamos resolviendo el problemas de las interfaces de usuarios definiendo una estructura arquitectónica para tal motivo. En un mismo sistema convivirán varias estructuras o soluciones de índole arquitectónico. En este caso solamente estamos tratando con el aspecto de la interacción hombre máquina. Las soluciones que están relacionadas con otros problemas arquitectónicos como de seguridad, concurrencia, el mapeo del modelo de objetos con bases de datos relacionales, el estado de las sesiones de usuarios en un

[4]También conocido como Model View Controller o simplemente MVC.

sistema, estrategias para sistemas distribuidos en forma remota, entre otros, estarán representadas cada una de ellas por estructuras arquitectónicas que resuelven cada problema.

2.8. Alcanzar la solución

MVC es un poderoso patrón compuesto por otros patrones que ayudan a definir la arquitectura de una aplicación. Para entender o aprender a aplicar el patrón MVC es necesario conocer otros patrones que se usan, ya no a un nivel de arquitectura de software, sino a un nivel de diseño o implementación.

Como los patrones son la clave para entender MVC, y los principios de diseño son la clave para entender los patrones, se comenzará primero introduciendo los conceptos fundamentales sobre los principios de diseño. Luego es necesario explicar los patrones básicos a nivel de implementación y diseño, y los principios en los que éstos patrones subyacen para lograr sus propósitos, para pasar así a explicar, con claridad, el patrón compuesto Modelo Vista Controlador.

Capítulo 3

Principios de diseño de Software

"Simple clear purpose and principles give rise to complex intelligent behavior. Complex rules and regulations give rise to simple stupid behavior."

Dee Hock[1]

En la búsqueda de los factores que afectan el proceso y resultado del diseño, a través del tiempo, y de la experiencia, se han propuesto un conjunto de axiomas o principios de diseño de software para lograr un diseño orientado a objetos flexible y reutilizable.

[1]Dee Hock, fue el fundador y CEO de VISA credit card association. Desarrolló el concepto de un sistema global para el intercambio electrónico de valores, que luego se convirtió en VISA.

3.1. Qué son los principios

Los principios no son reglas, leyes, ni doctrinas. Las reglas o leyes son fijadas por algún ente que de alguna forma consideramos superior o mantiene un dominio, el cual identifica, señala, e impone las reglas como algo muy importante como para dejarlo al libre juicio. Las reglas no requieren, ni se benefician de la interpretación de quién las sigue. No debemos romper las reglas si pretendemos seguirlas.

Una doctrina es un conjunto de instrucciones o enseñanzas que fundamentan el juicio, modo de actuar, y comportamiento. Es autoritativa pero algo flexible, suficientemente definida para fijar posiciones respecto a una materia o en situaciones específicas, y también lo bastante adaptable como para tratar situaciones variadas y diversas.

Los principios nos sirven como guías, por los cuales debemos esforzarnos en aplicarlos, más que adherirnos ciegamente y seguirlos todo el tiempo como si fuesen reglas. Necesitamos internalizar e incorporar los principios y tenerlos presentes en nuestras mentes al momento de diseñar, y así saber que cuando estamos violando un principio, es por buenas razones. Cada vez que no seguimos un principio debe haber un intercambio del cual salimos favorecidos.

Un principio es una norma o idea fundamental que rige el pensamiento o la conducta, en nuestro caso, sobre el diseño de software.

En la ética, los principios morales enuncian las cosas que el hombre ha descubierto que repercuten de una manera negativa para su vida y para la vida de los demás.

En el diseño de software, los diseñadores a través de la experiencia en la creación de sistemas informáticos complejos, y de la búsqueda en lograr sistemas reutilizables y flexibles, fueron descubriendo formas de diseñar que los acercaban a sus objetivos, y que luego fueron enunciados como principios de diseño.

Un diseño poco elaborado, o pobre, es un síntoma de que se está violando uno o más de los principios de diseño de software. Podemos diagnosticar los problemas que posee un diseño a través de los principios.

Los principios de diseño del software orientado a objeto ayudan a los desarrolladores a eliminar los síntomas de un diseño pobre y mal logrado, y alcanzar un mejor diseño para el conjunto de características que deseemos implementar.

Los principios son el producto ganado con el esfuerzo de décadas de experiencia en la ingeniería de software. No son el producto de una sola mente, pero representan la integración de los pensamientos y las escrituras de un gran número de desarrolladores

de software e investigadores. Aunque ellos sean presentados aquí como los principios del diseño orientado a a objetos, ellos son en realidad casos especiales de principios que perduraron en la ingeniería del software.

Debemos aplicar los principios sólo cuando sea necesario, o sea, cuando un diseño pobre produzca un conflicto. No debemos aplicarlos solo por el hecho de aplicarlos si es que no existe un problema, esto sería un error. Los principios están para ayudarnos a resolver conflictos, y aplicarlos sin la existencia de un conflicto, puede llevarnos a un sistema con una complejidad innecesaria.

3.1.1. Etimología

Del latín principium 'comienzo, primera parte' a su vez derivado de prim- 'primero, en primer lugar' y cap(i)- que aparece en el verbo capere 'tomar, coger, agarrar', por lo que literalmente principium es 'lo que se toma en primer lugar'. Se le puede llamar principio a los valores morales de una persona o grupo.

Etimológicamente el término latino principium está compuesto por la raíz derivada de pris, que significa «lo antiguo» y «lo valioso» y de la raíz cap que aparece en el verbo capere 'tomar, coger' y en el sustantivo caput 'cabeza'. Tiene, entonces, un sentido histórico («lo antiguo»), un sentido axiológico («lo valioso») y un sentido ontológico («cabeza»).[WIK]

Según el Diccionario de la Real Academia Española de la Lengua el término «principio» significa, entre otros, «punto que se considera como primero en una extensión o cosa», «base, origen, razón fundamental sobre la cual se procede discurriendo en cualquier materia», «causa, origen de algo», «cualquiera de las primeras proposiciones o verdades fundamentales por donde se empiezan a estudiar las ciencias o las artes».

3.2. Descripción de principios

Un principio en el que tal vez se apoya el paradigma orientado a objetos, es el de entereza conductual o *behavioural completeness*[2], el cuál establece que cada objeto de un sistema no solo conoce las propiedades del mundo real que representa, sino también conoce el modelo de comportamiento de la entidad que representa.

Esto no quiere decir que el objeto debe implementar cada comportamiento posible que podría llegar a necesitar. Quiere decir que todo el comportamiento asociado con el objeto

[2]Del libro Naked Objects[PM02] de Richard Pawson and Robert Matthews, capitulo "A critical look at object-orientation".

que es necesario para la aplicación que se está desarrollando debe ser una propiedad de ese objeto y no debe ser implementada en otro lugar de la aplicación.[PM02]

De alguna forma, este principio se puede considerar como nativo en el paradigma orientado a objetos. Cuando diseñamos un sistema orientado a objeto, ya pensamos en objetos completos conductualmente que van a ser instanciados y van a colaborar con otros objetos a través de mensajes para solucionar algún problema de la vida real.

Aun así, en el diseño orientado a objetos vamos a necesitar de otros principios que fueron elaborados durante algo mas de 40 años en la experiencia del uso de este paradigma para desarrollar sistemas.

3.2.1. Interfaces e implementación

Principio: "Programar para interfaces, no para una implementación".

Lo que realmente se quiere decir con programar para interfaces es programar para supertipos. Se está hablando del concepto de interfaz que una clase presenta. Cuando más abstracto y genérico el tipo, como lo es una clase abstracta o una interfaz de clase[3], el objeto actual concreto en tiempo de ejecución se encuentra referenciado dinámicamente y no estáticamente al código que lo representa.

La clave de este principio es el de aprovechar al máximo el polimorfismo programando para un supertipo, así cuando el tipo declarado de una variable es una clase abstracta o interfaz de clase, el objeto asignado a estas variables puede ser de cualquier implementación concreta del supertipo.

Cuando la herencia de clases es usada correctamente, todas las clases que derivan de una clase abstracta compartirán su interfaz. Al tratar los objetos sólo en término de la interfaz definida por una clase abstracta permite desacoplar los clientes con los tipos específicos de objetos que implementan esta interfaz, de esta manera los clientes desconocen las clases concretas reduciendo las dependencias de implementación en un sistema.

Este es un principio ampliamente usado en todos los patrones, trata acerca de las dependencia entre objetos teniendo en cuenta sus relaciones, las cuales deben ser cuidadosamente manejadas en un sistema, y nos dice que depender de interfaces es una dependencia sana y a la larga resulta beneficioso.

[3]Una interfaz de clase es una clase abstracta pura que no contiene ninguna implementación, sólo puede contener declaraciones de métodos, y constantes.

Una interfaz define el vocabulario de la colaboración entre objetos, una vez que entendemos el vocabulario, entendemos más sobre el sistema, porque al comprender todas las interfaces somos capaces de entender el vocabulario del problema[4].

Supongamos que disponemos de una jerarquía de clases con un supertipo abstracto `Figura` y clases de implementación derivadas como `Rectangulo`, `Triangulo` y `Cirulo`.

```
...
Triangulo t = new Triangulo();  //programando para una implementacion
Figura c = new Circulo();       //programando para una interfaz
Figura r = new Rectangulo();    //programando para una interfaz
...
```

Código 3.1: Programar para una interfaz.

3.2.2. Composición sobre herencia

Principio: "Favorecer la composición de objetos a la herencia de clases".

La herencia y la composición son dos técnicas de reutilización de código en sistemas orientados a objetos. La herencia toma la totalidad de otra implementación y la extiende para definir su propio tipo con un comportamiento más especializado, mientras que la composición nos permite tomar partes de diferentes implementaciones que son de interés para componer un nuevo tipo.

La herencia es una definición estática de comportamiento, y la composición define su comportamiento dinámicamente. Por esto, en una herencia de clases existe un fuerte acople entre la clase base y la clase derivada. En la composición este acople se reduce al conectar solo una parte de la implementación que queremos delegar a un objeto a través de una interfaz.

Los inconvenientes que tiene la herencia es que no es posible cambiar el comportamiento del objeto del tipo heredado en tiempo de ejecución, además las clases bases suelen definir parte de la representación física de las subclases, lo que expone en las clases derivadas la implementación de la clase base. Se dice que la herencia rompe la encapsulación.

La composición de objetos se define en tiempo de ejecución a través de objetos que se componen de referencias a otros objetos, por lo tanto se requiere que los objetos compuestos tengan presente las interfaces de los objetos a los cuales referencia. Es decir, los objetos compuestos solo dependen de las interfaces de los objetos componentes.

[4]Erich Gamma en una conversación el 6 Junio del 2005.

Crear un sistema usando composición nos da mucha más flexibilidad, no solo nos permite encapsular diferentes comportamientos en familias de clases, sino que también nos permite cambiar el comportamiento en tiempo de ejecución, siempre y cuando el objeto que estemos componiendo implemente la interfaz correcta.[FFBS04]

Un diseño basado en la composición de objetos tendrá más objetos, y el comportamiento del sistema dependerá de sus relaciones en vez de estar definido en una clase.[GHJV95]

3.2.3. Única responsabilidad

Principio: "Una clase debería tener sólo una razón para ser modificada."[5].

Si permitimos que una clase no solo se ocupe para lo que fue creada, sino que también cargue con más responsabilidades, le estamos dando a la clase más razones para ser modificada. En este principio, una razón para modificar una clase se define como una responsabilidad para la clase. Si podemos encontrar más de un motivo para modificar una clase, entonces la clase tiene más de una responsabilidad.

Supongamos una clase `Rectangulo` que tiene dos métodos, uno calcular su área y otro para dibujarse en una ventana. Una aplicación gráfica que dibuja rectángulos en una ventana, y un sistema geométrico que hace cálculos sobre la figura geométrica hacen uso de la clase `Rectangulo`.

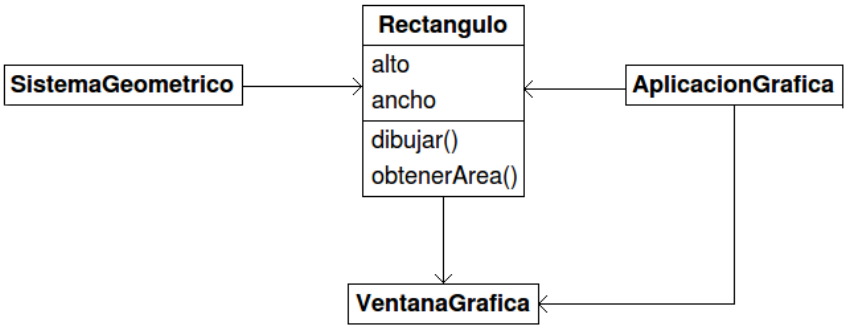

FIGURA 3.1: Clase `Rectangulo` con más de una responsabilidad.

Este primer diseño viola el principio de única responsabilidad, ya que `Rectangulo` tiene dos responsabilidades, o dos razones para cambiar. Posee la responsabilidad de proveer el modelo matemático geométrico del rectángulo, y además también carga con la responsabilidad de dibujar el rectángulo en una ventana gráfica. Un mejor diseño resulta

[5]Principio también conocido como "Single Responsibility Principle".

de separar estas dos responsabilidades en dos clases diferentes, como se muestra en la figura siguiente.

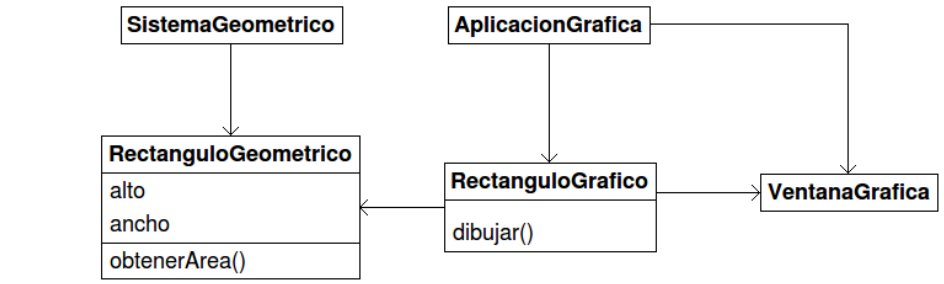

FIGURA 3.2: Aplicando el principio de única responsabilidad.

Se está hablando sobre cuán cohesiva debe ser una clase. Al darle a una clase una sola responsabilidad le estamos dando un alto grado de cohesión a la misma. Si una clase tiene más de una responsabilidad, estas responsabilidades se encontrarán altamente acopladas en una misma clase. De ser así, al realizar un cambio en el código que representa una responsabilidad puede provocar que la clase no cumpla como es requerido con la otra responsabilidad.

El principio de única responsabilidad es uno de los principios más simples, pero uno de los más difíciles de aplicar correctamente. Es altamente dependiente del contexto del dominio de la aplicación que estemos desarrollando. Se trata de realizar una clasificación adecuada de los conceptos del dominio del problema. Una razón para cambiar existe sólo si el cambio realmente puede ocurrir en el contexto que estamos trabajando. Si no lo consideramos de ésta forma, terminaríamos sobredimensionando el diseño con muchas más clases de las que realmente se necesitan, y agregaríamos una complejidad innecesaria.

3.2.4. Acoplamiento débil

Principio: "Esforzarse en lograr un diseño de acoplamiento débil entre objetos que interactúan"[6].

Con acoplamiento se hace referencia a una conexión o relación entre dos o más partes, y encontramos como medida del acople el nivel de dependencia existente entre estas partes. Cuando hay una gran dependencia entre varias partes, podemos decir que estas partes se encuentran fuertemente acopladas.

Este principio recomienda la creación de un tipo específico de relación basado en el conocimiento mínimo y necesario de las partes para que la relación funcione correctamente

[6]Este principio es conocido como "Loose Coupling Principle".

y sin complicaciones. No es necesario tener un conocimiento completo de un objeto para interactuar con el mismo de modo de lograr lo que se necesita.

Las relaciones entre clases deben estar bien definidas y deben ser restringidas. Las interfaces o clases abstractas ayudan a mantener las relaciones tanto bien definidas como restringidas, porque tanto las interfaces como las clases abstractas restringen los detalles de información que un objeto puede brindar como interfaz. Debemos enfocarnos sólo en los elementos requeridos de la clase en el momento de una solicitud o petición a un objeto.

Las peticiones solo deben ser manejadas por objetos concernientes en todo lo necesario e implicado en realizar la petición, ni más, ni menos. Al instanciar clases que sólo se ocupen de la cuestiones relacionadas con la petición que se necesita en ese momento, logramos un diseño débilmente acoplado.

Con clases débilmente acopladas podremos modificar implementaciones con mayor libertad y sin grandes cambios en un sistema debido a la baja dependencia entre ellas.

En un sistema podemos reducir el acoplamiento, pero nunca eliminarlo. El acoplamiento indica el grado de dependencia que existe entre las partes de un sistema. Naturalmente cada una de las partes de un sistema va a depender de alguna otra parte para realizar sus respectivas funciones, pero lo que realmente preocupa es cuándo una parte depende de la implementación de otra.

Al acoplamiento entre partes también es posible verlo a un nivel de mayor abstracción que una clase, como entre módulos o partes del sistema más grandes que una clase. Se pueden encontrar diferentes niveles de acople entre las partes o módulos de un sistema, a continuación se muestra un listado de estos niveles, yendo de fuertemente a débilmente acoplado:[C2W] [7].

- Acople de contenido (peor):
  Cuando un módulo usa y altera datos en otro módulo.
- Acople de control:
  Módulos comunicados a través de banderas de control.
- Dato global común:
  Módulos comunicados vía datos globales.
- Acople de parámetros:
  Comunicación vía estructuras de datos u objetos pasados como parámetros, donde la estructura contiene más información de la necesitada.

[7] Niveles enunciados en http://c2.com/cgi/wiki?CouplingAndCohesion.

- Acople de datos (mejor):
  Comunicación a través de parámetros, pero pasando solamente la información necesaria, o escondiendo detalles a través de una interfaz.

- Sin acople:
  Módulo totalmente independiente, aislado.

Éstos niveles de acople surgieron en el contexto de la programación procedimental estructurada, pero se aplica también a otros paradigmas. El último nivel del listado carece de sentido en un sistema real, es teórico, ya que es totalmente impensable que existan módulos totalmente aislados cuando la idea es que existan una comunicación entre ellos para lograr una tarea.

3.2.5. Apertura-Clausura

Principio: "Los elementos constructivos del software como clases, módulos, funciones, deben estar abiertos para la extensión, pero cerrados para la modificación"[8].

Lo que este principio aconseja es que el diseño de un sistema debe estar preparado para que futuros cambios no sean sinónimos de una modificación en el código de los elementos constructivos establecidos que componen la aplicación. Si este principio es bien aplicado, los cambios al sistema serán alcanzados en su mayoría agregando nuevos elementos constructivos, no modificando el código existente que ya funciona correctamente.

En el paradigma orientado a objeto diríamos que: "una clase debe estar abierta para la extensión, pero cerrada para la modificación"[9]. Con este principio se pretende que nunca debería existir la necesidad de modificar el código en las clases existentes, donde toda nueva funcionalidad debería poder ser agregada, creando nuevas clases o métodos, o reutilizando código existente.

Para lograr esto, debemos identificar los aspectos de la aplicación que varían, encapsularlos y separarlos del resto del código que permanece sin variar. De esta manera, podremos alterar el sistema extendiendo de las partes que no varían, agregando nuevas partes, o modificando sólo las partes aisladas propensas al cambio, en lo posible, sin afectar la parte resuelta que puede permanecer sin cambio.

Que una clase sea abierta para la extensión significa que su comportamiento puede ser extendido o heredado para satisfacer los nuevos requerimientos de la aplicación. Una

[8] Principio conocido como Open-Closed Principle.
[9] Se le atribuye este principio a Bertrand Meyer, diseñador inicial del lenguaje Eiffel.

clase cerrada para la modificación no debe, en principio, contener comportamiento que sea factible de cambio según los requerimientos. Se entiende que al extender o heredar el comportamiento de una clase no resulta en cambios en el código fuente de la clase base.

La clave para la aplicación de éste principio es la abstracción adecuada e inteligente de los supertipos. Es posible crear métodos abstractos, clases abstractas, o interfaces de clases que representen un grupo de comportamientos ilimitados, y en las subclases derivadas implementar el comportamiento del sistema que es tendencioso al cambio. De esta forma, se puede encapsular y separar lo que varía dentro de estas subclases, y lograr mantener cerrado para la modificación las partes del sistema que permanecen fijas y solo dependen de las abstracciones.

Adherirnos al principio apertura-clausura nos lleva a sistemas flexibles, reusables, y mantenibles. Lo que plantea este principio parece inalcanzable e ideal, ya que no se puede pretender aplicar este principio a todas las partes del sistema porque resultaría en un exceso de complejidad. Los desarrolladores deben dedicarse a conocer el dominio de la aplicación y estimar qué partes son más propensas al cambio para aplicar este principio.

3.2.6. Sustitución de Liskov

Principio: "Los subtipos deben ser sustituibles por su tipo base"[10].

Lo que se busca con este principio es algo como la siguiente propiedad de sustitución: "Si por cada objeto o_1 del tipo S hay un objeto o_2 del tipo T tal que todos los programas P definidos en términos de T, el comportamiento de P no cambia cuando o_1 es sustituido por o_2, luego S es un subtipo de T"[11].

Es muy importante tener en cuenta que para cumplir con este principio es necesario que exista una equivalencia de comportamiento para que un subtipo sea sustituible por su tipo base. Si una clase B hereda de una clase A, donde sea que usemos B deberíamos poder usar A para cumplir con este principio.

Los métodos que usan referencias a clases bases deben ser capaces de usar objetos de clases derivadas sin saberlo. Por ejemplo, dada una clase con un comportamiento y alguna posible subclase que puede implementar el comportamiento original, quien espera una instancia de clase base no debe sorprenderse de nada si ésta es sustituida por una instancia de subclase.

[10] Principio es conocido como "Liskov Sustitution Principle".

[11] Barbara Liskov, Data Abstraction and Hierarchy, SIGPLAN Notices, 23,5 (May, 1988). Special Interest Group on Programming Languages (http://www.sigplan.org).

Violar este principio nos ayuda a entenderlo. Supongamos que queremos modelar un cuadrado y un rectángulo. Sabemos que un cuadrado es un rectángulo, pero no todo rectángulo es un cuadrado, entonces podríamos proceder de la siguiente forma para modelar esta situación:

```
public class Rectangulo {
    public int ancho;
    public int alto;
    public void establecerAlto(int alto) {
        this.alto = alto;
    }
    public void establecerAncho(int ancho) {
        this.ancho = ancho;
    }
    public int calcularArea() {
        return ancho * alto;
    }
}
```

Código 3.2: Clase Rectangulo.

Al momento de modelar el cuadrado tenemos que tener en cuenta la restricción que esta abstracción debe poseer: en un cuadrado sus lados son iguales.

```
public class Cuadrado extends Rectangulo {
    public void establecerAlto(int alto) {
        this.alto = alto;
        this.ancho = alto;
    }
    public void establecerAncho(int ancho) {
        this.ancho = ancho;
        this.alto = ancho;
    }
}
```

Código 3.3: Clase Rectangulo.

Modelando de esta forma el rectángulo y el cuadrado se esta dejando de lado el principio de sustitución de Liskov, ya que la subclase `Cuadrado` no tiene un comportamiento equivalente al de su clase base `Rectangulo`.

Podemos decir que el modelo que se creó es consistente consigo mismo, pero no es consistente para todos los posibles usuarios del modelo, como se muestra en el siguiente método de alguna clase:

```
...
public void metodo(Rectangulo r) {
    r.establecerAncho(5);
    r.establecerAlto(4);
    assert (r.calcularArea() == 20);
}
...
```

Código 3.4: Violación del Principio de Sustitución de Liskov.

Este método establece el ancho y alto de lo que supone un rectángulo. El método funciona bien para rectángulos, pero ocasiona un error en `assert` al pasar un cuadrado. Este método expone la violación al Principio de Sustitución de Liskov.

El programador de `metodo(Rectangulo r)` asumió razonablemente que su método trabajaría con rectángulos, el problema fue asumir que un cuadrado es un rectángulo cuando trabajamos con objetos. Un objeto `Cuadrado` definitivamente no es un objeto `Rectangulo`, porque el comportamiento de un objeto `Cuadrado` no es consistente con el comportamiento de un objeto `Rectangulo`.

3.2.7. Separación de intereses

Principio: "Identificar y separar los diferentes tipos de intereses en un problema"[12].

Es un principio clave en la ingeniería de software y nos dice que en el dominio de un problema aparecen diferentes tipos de intereses, los cuales deben ser identificados y separados para manejar la complejidad y alcanzar orden, para lograr un sistema robusto que sea reutilizable, extensible, portable, mantenible y escalable.

Este principio puede ser aplicado de varias formas, y en varios niveles de abstracción, desde problemas de diseño de implementación, hasta problemas de diseño arquitectónico donde es más común que sea nombrado. Con este principio se afirma que los elementos de un sistema deberían tener exclusividad y singularidad de propósito.

Alcanzamos la separación de intereses estableciendo los límites para separar las diferentes partes en que se organiza un sistema y determinar un conjunto de responsabilidades para cada parte. Las responsabilidades no deben repetirse en ninguna de las otras partes del sistema para lograr que la separación sea altamente cohesiva y débilmente acoplada.

La esencia de la separación de intereses es lograr orden, donde cada parte delimitada debe interpretar un rol valioso y único para el sistema.

Aplicar este principio en el diseño de software puede resultar en valiosas ventajas:

[12] Principio conocido como "Separation of Concerns".

- Partes bien delimitadas con un propósito bien definido y la falta de duplicación de responsabilidades resulta en un sistema más fácil de mantener.
- Al incrementarse la mantenibilidad del sistema, éste a su vez es un sistema más estable.
- Lograr partes con un alto grado de cohesión y con responsabilidades que no se repitan, resulta en un sistema más fácil de extender.
- El bajo grado de acople facilita la reutilización de las partes en diferentes contextos.

Aplicar de manera correcta el principio de separación de intereses en un sistema ayuda a la rápida resolución de problemas porque la identificación de estos termina siendo más sencilla. Al resultar en un sistema bien organizado, su testeabilidad también es más sencilla.

Capítulo 4

Patrones de Diseño de Software

> *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."*
>
> Christopher Alexander[1]

Los principios de diseño de software son ideas generales sobre cómo diseñar para obtener un diseño exitoso. Estos principios se aplican en el diseño de todo tipo de sistemas de información, sin importar lo que estemos desarrollando, mientras que la aplicación de patrones persiguen una intención específica y bien definida, apuntan a cómo se resuelve un problema en particular que es conocido y recurrente.

Los patrones pueden llegar a ser muy complejos, los principios no. Pero aplicar los principios de diseño correctamente es difícil. La ventaja al aplicar los patrones es que estamos aplicando al mismo tiempo los principios de manera correcta y probada. Es por esto que se puede considerar a los patrones como una herramienta para vencer de algún modo esta dificultad. Aun así, para entender los patrones, poder discutir sobre ellos, y adecuarlos a nuestro contexto, es necesario conocer los principios y tenerlos siempre presentes. Los patrones son herramientas muy poderosas, porque a partir de ellos podemos crear un vocabulario con un alto grado de abstracción, lo cual nos permite razonar de un modo más eficiente.

[1] Christopher Alexander es un reconocido arquitecto y es unos de los pioneros en el estudios de patrones relacionados con la arquitectura edilicia.[AIS+77]

4.1. Qué es un patrón

Un patrón es algo que sucede en forma repetida debido a fuerzas existentes, que podemos o no conocer, las cuales rigen y guían el suceso a la repetición.

En la arquitectura edilicia, uno de los pioneros en tratar el estudio de los patrones fue Christopher Alexander. Define que un patrón es una regla de tres parte que expresa la relación entre un cierto *contexto*, un *problema*, y una *solución*. Además, Alexander, muestra al patrón como un elemento en el mundo que está en relación con un cierto contexto, un cierto sistema de fuerzas que ocurren repetidamente en ese contexto, y una cierta configuración espacial que permite que esas fuerzas se resuelvan entre ellas. También pone al patrón como un elemento de lenguaje, donde el patrón es una instrucción, que muestra cómo esa configuración espacial puede ser usada, una y otra vez, para resolver el sistema de fuerzas dado. Alexander resume que, un patrón es al mismo tiempo una cosa, que ocurre en el mundo, y la regla que nos dice cómo crear esa cosa y cuándo debemos crearla. Es tanto un proceso y una cosa, tanto una descripción de una cosa que está viva, y una descripción de una proceso que generará esa cosa.[Ale79]

Cuando cualquier profesional trabaja en un problema es usual que tome como referencia algún trabajo que haya realizado con anterioridad para resolver algo similar, y use la esencia de la solución anterior para resolver el nuevo problema. Este pensamiento problema-solución es muy común en expertos de diferentes disciplinas, como arquitectura, economía, software, etc, y está relacionado con la detección de un patrón.

En las geociencias estudian, entre otras cosas, el clima de la tierra donde tratan de encontrar un patrón de comportamiento en las fluctuaciones del clima. Al ser tan grande la cantidad de fuerzas que pueden llevar a un suceso climatológico a algún tipo de repetición, se hace muy complejo explicar o entender un patrón que describa esta repetición. El clima es el resultado de una configuración espacial que permite que todas estas fuerzas actuantes se resuelvan entre ellas. Encontrar cómo es esta configuración espacial es en parte encontrar un patrón para tal comportamiento.

Como personas sociales, siempre utilizamos patrones para resolver problemas, aunque tal vez no nos damos cuenta. Todas nuestras soluciones responden a un patrón porque somos individuos culturalizados, y esta cultura actúa de alguna forma como un principio. Hay muchos comportamientos que responden a patrones conocidos con los cuales podemos alcanzar el éxito en diferentes áreas, y hay otros que faltan descubrir, para que su descripción y posterior aplicación nos impulse a obtener mejores soluciones.

En la ingeniara de software existen problemas de diseño que aparecen repetidamente, que se encuentran en contextos similares, donde cada problema se relaciona fuertemente

con determinados principios de diseño que actúan como un sistema de fuerzas que llevan a que las soluciones a estos problemas se desarrollen de una manera parecida, y a la larga terminen definiendo patrones de diseños útiles para resolver futuros problemas en condiciones similares.

El estudio de los patrones nos ayudan a construir sistemas sobre la experiencia colectiva de ingenieros de software expertos. Los patrones capturan la experiencia exitosa existente en el desarrollo de software y ayudan a promover buenas prácticas de diseño.

El pensamiento análogo es aquel que detecta un patrón en algo más conocido, común o cotidiano, con respecto a lo que se estaba pensando al momento de descubrir la analogía, al mismo tiempo refuerza y vigoriza la idea pensada. Las analogías conectan lo conocido con lo desconocido a través de algo en común, y de algún modo se relacionan con la detección de un patrón.

Los patrones no se crean, sino se descubren. Un patrón no es algo radical, y menos aún algo novedoso o innovador. Lo realmente útil de un patrón es su descripción, concebida a través del descubrimiento, que va a servir para potenciar la comunicación, el aprendizaje, y el pensamiento.

4.1.1. Etimología

Etimológicamente, la palabra patrón viene del latín *patronus*, que quiere decir protector, y éste a su vez viene de *pater* que quiere decir padre.[WIK]

El patrón responde de alguna forma a la figura de protector, como lo es un padre o una madre, alguien a quien seguir los pasos, que marca una guía, debido a que las acciones del protector favorecen y protegen la supervivencia del protegido. Es el primer contacto con la experiencia misma capturada en la figura de una persona y expresada con acciones.

4.2. Características de un patrón

Para esclarecer el concepto de patrón, se expondrán las características presentes en un patrón en el área del desarrollo de software.

- Un patrón aborda un problema de diseño recurrente.
  En el tema de tesis que se está tratando, el problema de las interfaces de usuario aparece cuando desarrollamos un sistema interactivo hombre computador. Esta interactividad denota una situación de diseño específica, donde el patrón MVC

presenta una solución. Entonces, un patrón brinda una solución a un problema de diseño recurrente que surge en situaciones de diseño específicas.

- Los patrones documentan experiencias de diseño.
  Los patrones no son inventados o creados de la nada, son descubiertos a través de experiencias ya probadas exitosamente. Proveen un medio para la reutilización del conocimiento y experiencia de los que ya pasaron por los problemas que estamos tratando de solucionar. Con la descripción de los patrones, el conocimiento no sólo existe en las cabezas de unos cuantos expertos, sino que está disponible para todos.

- Los patrones identifican y especifican abstracciones.
  Un patrón comúnmente describe varios componentes, clases, u objetos, detallando sus responsabilidades, cómo se interrelacionan, y cómo cooperan entre ellos. Todos estos componentes en conjunto resuelven el problema que el patrón trata. Un patrón de diseño posee un nivel de abstracción más alto que una simple clase o instancia.

- Los patrones proveen un vocabulario común.
  Algo muy importante de un patrón es su nombre, que si es elegido cuidadosamente se puede convertir en parte de un lenguaje de diseño. Cuando nos comunicamos usando patrones con otro desarrollador, a través del nombre del patrón estamos comunicando un conjunto de características, propiedades, y restricciones que el patrón representa. Los patrones nos permiten decir más con menos. Con solo el nombre de un patrón estamos caracterizando tanto el problema como la solución, y nos evitamos una larga y compleja descripción. Hablar a nivel de patrones nos permite mantener una discusión más tiempo en el nivel de diseño, y no desviarnos en detalles técnicos entorpecedores de la implementación. Esto también favorece a la agilidad del desarrollo del sistema.

- Los patrones son un medio para documentar.
  Permiten describir la visión que se tiene en mente al momento de diseñar. Con patrones se puede expresar de una manera más concisa la arquitectura de software de una aplicación, y evita que se viole esta visión de diseño al realizar la implementación, o en momento de realizar modificaciones en el código.

- Los patrones ayudan a construir arquitecturas.
  Como cada patrón provee un conjunto predefinido de componentes, roles, y relaciones, esto puede ser usado para especificar aspectos particulares de la estructura del software. Los patrones actúan como bloques constructivos para la construcción de arquitecturas complejas y heterogéneas. Entendiendo y aplicando patrones ahorramos más tiempo que cuando buscamos nuestras propias soluciones. Aun

así, aunque un patrón determine la estructura básica de la solución a un problema particular de diseño, no especifica la solución detallada y completa de una arquitectura.

- Los patrones ayudan a manejar la complejidad del software.
  Cada patrón describe una forma de solución probada y destilada a través del tiempo y experiencia, especificando los componentes que necesitamos, los roles que deben cumplir las partes, los detalles que deben ser ocultos, las abstracciones que deben ser visibles, y cómo todas estas partes deben trabajar en conjunto. Cuando encontramos que una situación concreta de diseño puede ser resuelta por un patrón, no vale la pena encontrar una nueva forma de solucionar el problema. Si aplicamos de manera correcta un patrón, se puede confiar en la solución que el patrón nos brinda. El manejo de un lenguaje de patrones como elementos abstractos de pensamiento potencian nuestras mentes para vencer la complejidad.

- Los patrones son trabajos en progreso.
  Los patrones que son fructíferos para la producción de software deben estar lo suficientemente completos y maduros. Es la experiencia práctica ganada a través del tiempo. Un patrón evolucionan sobre la experiencia en el desarrollo de sistemas específicos donde definen su contexto, está sujeto a continuas revisiones, mejoras, refinamientos, y se desea de que su descripción sea lo más completa posible.

4.3. Esquema de un patrón

El esquema de un patrón, como un todo, denota o pone en manifiesto un tipo de instrucción o guía a seguir que establece la relación entre un contexto dado, un cierto problema que aparece en ese contexto, y una apropiada solución al problema. Estas tres partes del esquema de un patrón se hallan relacionadas, y son las características básicas que debemos identificar para detectar el suceso de un patrón. No puede dejar de existir ningunas de estas partes para que se dé un patrón, estas partes son lo que hacen a un patrón.

Para el éxito de un patrón su propia comunicación es esencial, por lo tanto una presentación y descripción apropiada es primordial para comunicarlo exitosamente.

- Contexto
  El contexto describe situaciones comunes que dan origen a un problema que aparece en forma repetidas, estas son las situaciones en la cuales se puede aplicar un patrón determinado. Son situaciones comunes que afrontaron varios diseñadores expertos, los cuales fueron descubriendo los patrones a través de la experiencia y las buenas

prácticas. Especificar el contexto de un patrón puede ser difícil, es casi imposible determinar todas las situaciones en las cuales el patrón pueda ser aplicado. Una forma más pragmática de expresar el contexto de un patrón es listar todas las situaciones conocidas en donde aparezca el problema que pueda ser solucionado por un patrón en particular. Esto no garantiza que cubramos cada situación en que un patrón pueda ser relevante, pero al menos nos da una buena pista. Un patrón no es una regla, ni un principio general, por lo que el contexto en donde un patrón se aplica debe ser claramente identificado.

- Problema
  Esta parte describe el problema específico que estamos teniendo, que aparece una y otra vez en un contexto dado, por el que otros diseñadores ya pasaron. Comienza con una especificación general del problema tratando de capturar la esencia del mismo, y dejando en claro cuál es la cuestión concreta de diseño que debemos resolver. El problema ocurre cuando no hay una solución fácilmente aplicable, esto sucede muchas veces porque la solución debe cumplir con requisitos y restricciones que actúan como fuerzas, a veces contradictorias, que generan el conflicto.
  En la descripción del problema hay que hacer notar y describir estas fuerzas que son las que nos guiarán a una solución, tanto las metas como las restricciones que la solución debe cumplir. Estas fuerzas, como se dijo, algunas veces pueden ser contradictorias entre sí, son las que nos ayudan a entender en detalle el problema. Cuanto más detalladas sean las metas y restricciones que crean el conflicto, más detallado será el problema.

- Solución
  La solución que representa la aplicación de un patrón debe mostrar cómo resolver las fuerzas asociadas con el problema. Un patrón especifica una cierta estructura, o configuración espacial de los elementos, que aborda los aspectos estáticos y dinámicos de la solución. De la parte estática, la estructura describe sus componentes, la relación entre ellos, y con sus responsabilidades. También, cada patrón especifica el comportamiento en tiempo de ejecución de estos componentes, o los aspectos dinámicos de la solución, describiendo cómo estas partes colaboran, cómo el trabajo es organizado, y cómo se comunican entre ellos. La solución en definitiva describe los elementos que constituyen el diseño, sus relaciones, responsabilidades, y colaboraciones. Los patrones no proveen un elemento concreto para aplicar, sino un modelo abstracto ya probado de la solución, que puede aplicarse en diferentes situaciones, y su aplicación resultará en una solución a medida del problema específico que estemos tratando.

4.4. Descripción de un patrón

La descripción de un patrón es algo muy importante porque una de las principales metas es su comunicación y la transferencia de la experiencia de lo ya probado.

Varios autores proponen diferentes formas de descripción de patrones, pero aun así todas las formas que presentan contienen elementos comunes que no pueden dejarse de lado. El propósito de una descripción es comunicar un patrón de manera completa para que el lector pueda lograr aplicarlo en una de las infinitas variaciones que un proyecto de software puede tener. Describir un patrón de forma completa no significa describir su aplicabilidad en forma exhaustiva, ya que esto sería imposible, significa dejar en claro el contexto, el problema, y la solución.

Cuadro 4.1: Elementos Descriptivos de un patrón en Diferentes Autores

GoF[GHJV95]	Buschmann et al.[BMR+96]	Fowler et al.[Fow02]
Nombre y Clasificación	Nombre	Nombre
Propósito	Conocido como	Intensión y esquema
También conocido como	Ejemplo	Problema motivante
Motivación	Contexto	Como funciona
Aplicabilidad	Problema	Cuando usarlo
Estructura	Solución	Leyendo aún más
Participantes	Estructura	
Colaboraciones	Dinámica	
Consecuencias	implementación	
implementación	Ejemplo Resuelto	
Código de ejemplo	Variantes	
Usos conocidos	Usos Conocidos	
Patrones relacionados	Consecuencias	
	Ver también	

Unos de los elementos que describe un patrón es su nombre, el cual es crucial porque parte del propósito de los patrones es el de crear un vocabulario común para una comunicación más efectiva y eficiente.

Los diferentes esquemas de descripción podrán ser más o menos estructurados, pero todos estos elementos descriptivos surgen del contexto, problema, y solución de un patrón. La descripción de los patrones que se dará en esta tesis tratará de ser más libre, haciendo hincapié en denotar directamente elementos más generales como el contexto, problema, y solución, sin dejar de lado cuestiones técnicas esclarecedoras bajo el paradigma orientado a objetos, como la estructura y dinámica de un patrón, y un ejemplo particular de aplicación cuando sea necesario.

4.5. Clasificación de patrones

Existen varios criterios de clasificación para la organización de los patrones en familias o grupos relacionados con propiedades comunes para la creación de un catálogo. La dimensión de un esquema de clasificación va a depender de la cantidad de criterios adoptados al momento de clasificar. Por ejemplo, un esquema bidimensional usa dos criterios en el proceso de clasificación. Cuando más dimensiones posea un esquema se logra una clasificación de patrones en grupos con características más específicas, pero hace más compleja y ambigua su selección.

El principal objetivo de una clasificación es la de facilitar la tarea de selección a los desarrolladores para alcanzar el patrón correcto a aplicar. Para que un esquema de clasificación sea eficiente, se piensa que un patrón no debe aparecer en más de dos grupos o familias de patrones.

Los criterios por los que comúnmente se clasifican los patrones son:

- Disciplina
  Disciplina, área, o campo donde los patrones se aplican, como ingeniería de software, economía, urbanismo, bioquímica, sociología, etc. Si bien un patrón en sí mismo es una abstracción, es muy raro un patrón que se aplique a una disciplina sea relevante en otra.

- Dominio
  En el área de la ingeniería del software, los patrones han sido aplicados en distintos dominios, como: sistemas de tiempo real, comunicaciones, sistemas distribuidos, interfaces de usuario, etc.

- Paradigma
  Los patrones alcanzaron gran popularidad en el paradigma orientado a objetos, pero también existen patrones en diferentes paradigmas, como en el paradigma procedimental, funcional, lógico, etc.

- Granularidad
  Los patrones de software varían en su granularidad y nivel de abstracción. Hay patrones que se pueden apreciar solo en altos niveles de abstracción como los patrones arquitectónicos. Al elegir este criterio puede darse el caso que un mismo patrón pueda ser clasificado en diferentes categorías de granularidad, aun así este criterio resulta beneficioso incluirlo en un esquema de clasificación.

- Propósito

  Este criterio de clasificación permite separar los patrones según la situación concreta donde se pueden aplicar. Representa el tipo de problema que el patrón resuelve y refleja lo que hace un patrón en particular. Todos los esquemas conocidos de clasificación incluyen este criterio, aunque con diferentes grados de generalidad. Por ejemplo, GoF[2] proponen tres clases de propósitos: de creación, estructurales y de comportamiento en [GHJV95], mientras que Buschmman menciona clases más específicas de propósitos y les da el nombre de categoría de problemas, donde hace una separación de clases más fina: comunicación, control de acceso, descomposición estructural, organización de trabajo, etc, en [BMR+96].

- Ámbito

  Este criterio sirve para clasificar patrones según características de implementación. En el paradigma orientado a objetos, los patrones pueden ser implementados usando relaciones entre clases y subclases a través de la herencia determinada de manera estática, o por composición de objetos donde las relaciones entre objetos pueden ser cambiadas en tiempo de ejecución. Estas dos clases de ámbitos son llamadas ámbito de clase y ámbito de objetos respectivamente en [GHJV95].

La clasificación que adoptemos va a depender de lo que queramos resaltar o mostrar según en lo que estemos trabajando. Como se quiere demostrar lo beneficioso que resulta la aplicación del patrón MVC en el contexto del desarrollo de software interactivo, se usará como criterio la **granularidad** para lograr una clasificación que resalte los diferentes niveles de abstracción y los distintos rangos de escalas en los patrones. Esto será de utilidad para diferenciar los patrones en lo que su aplicación resulta en una estructura que define una arquitectura en un sistema, de otros patrones que se aplican a elementos de implementación más finos.

Otro criterio que se incluirá en el esquema de clasificación que usaremos, es el criterio de **propósito**. Es tal vez el criterio más provechoso para categorizar los patrones, y es usado en casi todas las clasificaciones presentadas por distintos autores. Es la clasificación según la intención o propósito que el patrón persigue. Esta intención, que existe en un patrón, representa el tipo de problema que resuelve, y describe la situación concreta donde el patrón puede ser aplicado.

Además, es claro que se estarán usando de manera tácita el criterio de disciplina y de paradigma. Donde la disciplina o campo que nos concierne es el de la ingeniería de software, y el paradigma orientado a objetos el modelo donde radican los elemento que construyen el software.

[2] Con GoF "Gung of Four" nos referimos a los autores del libro "Patrones de Diseño".[GHJV95]

Así llegamos a un esquema bidimensional donde hay sólo dos criterios de clasificación: el criterio de granularidad, y el criterio de propósito. Esta es la clasificación propuesta por Buschmann en [BMR+96].

Resulta importante determinar la clasificación según el criterio de granularidad para determinar la escala de abstracción que puede poseer un patrón:

- Patrones de Implementación
  También son llamados modismo[3] o patrones de codificación. Estos patrones tratan con la implementación de problemas de diseño resueltos. Un patrón de implementación es un patrón con un bajo nivel de abstracción con respecto al software en general, está directamente relacionado con la programación, y tiene como objetivo lograr una codificación legible y entendible para otras personas. Describen cómo implementar aspectos particulares de componentes, o de la relación entre ellos usando las características de un lenguaje de programación dado. La mayoría de estos patrones son específicos a un lenguaje, y capturan la experiencia en el uso de estos lenguajes en la programación, son hábitos de programación que resultan en un código legible y entendible. En ocasiones los mismos patrones tienen apariencias diferentes en distintos lenguajes de programación, y algunas veces, algunos patrones que son útiles en algunos lenguajes de programación, no tienen sentido en otros.

  Por ejemplo, una operación crítica en C++ es la asignación de referencias. El patrón llamado "Counted Body", también conocido como "Reference Counting", sirve para manejar recursos alojados dinámicamente en memoria. En Smalltalk, o Java, lenguajes que ya proveen el mecanismo de "Garbage Collection", este patrón no tiene sentido.

- Patrones de Diseño
  Son los patrones en los cuales sus estructuras no organizan los demás elementos del sistema, como lo harían las estructuras arquitectónicas descriptas en el Capítulo 2. Las subestructuras de una arquitectura de software, como también las relaciones que existan entre ellas, usualmente consisten en varias unidades de arquitecturas más pequeñas donde estos patrones de diseño pueden ser aplicados. Un patrón de diseño provee un esquema que describe estas subestructuras o las relaciones entre ellas, que son comunes y recurrentes en un sistema de software. La aplicación de patrones de diseño no tiene efecto en la arquitectura fundamental del sistema de software, pero puede tener una fuerte influencia en las estructuras de los subsistemas. Son patrones con un nivel de abstracción medio, pero con un nivel de

[3] Denominados como 'Idioms' en [BMR+96] e 'Implementation Patterns' en [Bec07].

abstracción mayor que los patrones de implementación. Tienden a ser independientes a un lenguaje de un programación específico, pero a veces algunos patrones de este tipo pueden no tener sentido en diferentes paradigmas.

Por ejemplo, si estamos trabajando con lenguajes bajo el paradigma procedimental, podríamos hablar de patrones como herencia, encapsulación, y polimorfismo, lo cual no tiene sentido en términos de un lenguaje orientado a objetos, porque estos patrones se encuentran incluidos en forma primitiva en el paradigma que implementan. Podríamos usar el patrón encapsulación, y aplicado mediante la implementación de los tipos abstractos de datos o ADT (Abstract Data Type) en los lenguajes procedimentales, pero no tiene sentido hablar de este patrón en el paradigma orientado a objeto, donde la encapsulación, no es solo una característica nativa del paradigma, sino también uno de sus pilares, como lo son la herencia y el polimorfismo.

- Patrones Arquitectónicos
  Estos patrones son los que contienen el más alto nivel de abstracción. Su aplicación resulta en las estructuras fundamentales con las que podemos describir la arquitectura de un sistema.

  Un patrón arquitectónico expresa un esquema de organización estructural fundamental para sistemas de software. Provee un conjunto de subsistemas predefinidos, especifica sus responsabilidades, e incluye reglas y pautas para organizar las relaciones entre ellos. [BMR+96]

Los patrones arquitectónicos son plantillas mentales para elaborar arquitecturas de software concretas. Especifican las propiedades estructurales de un aspecto de un problema al cual el sistema debe brindar solución, y su aplicación tiene un impacto a su vez en la arquitectura de sus subsistemas. La selección de un patrón arquitectónico es por lo tanto una decisión de diseño fundamental cuando desarrollamos software y debe ser tomada a conciencia porque influenciará en varios aspectos en el desarrollo del mismo.

El patrón Modelo Vista Controlador es uno de los patrones de arquitectura más conocidos, y provee una estructura organizativa para sistemas interactivos.

Capítulo 5

Patrones y principios en torno a MVC

El patrón Modelo Vista Controlador está compuesto de otros patrones para lograr su propósito y resolver los requerimientos y restricciones que impone el contexto de las aplicaciones interactivas. Son varios los patrones que aparecen en torno a MVC, y cada uno de ellos surge por diferentes necesidades que plantean los sistemas interactivos. Estos patrones pueden pertenecen a la familia de los patrones de diseño e implementación, que en este caso se encontrarán en una estrecha relación de colaboración, y tendrán entre todos una misma intención: la de resolver el problema de la interactividad entre usuario y sistema.

Entender el patrón arquitectónico MVC en su forma clásica para poder realizar una aplicación del mismo resulta, entre otras cosas, de entender los patrones que lo componen.

5.1. Patrón Strategy

Define una familia de algoritmos o comportamientos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes u objetos que lo usan.

5.1.1. Otros nombres

Este patrón también es conocido como *Policy*.

5.1.2. Contexto

Objetos de un mismo tipo con diferentes comportamientos intercambiables.

5.1.3. Problema

A veces es necesario definir objetos de un mismo tipo, que dependiendo de diferentes circunstancias se comporten de diferentes formas. Esto nos podría llevar a definir los distintos comportamientos en una misma clase, e implementarlos dentro de múltiples sentencias de decisiones que verifiquen una determinada condición. Esta clase, así implementada, es propensa a ser modificada cada vez que aparezcan nuevas condiciones o nuevos comportamientos a implementar.

Otra forma sería plantear una herencia de clases que permita una variedad de comportamientos para un mismo tipo de objeto, pero este objeto estaría ligado estructuralmente a este comportamiento, y no permitiría modificar dinámicamente su comportamiento.

Si el comportamiento de un objeto está basado o depende de datos que no debería conocer, no estaría bien implementar ese comportamiento como parte de la clase que instancia esos objetos.

El problema se hace visible al establecer los diferentes requerimientos y restricciones que debe poseer la solución:

- Muchos objetos relacionados difieren solamente en su comportamiento, o alguna parte de su comportamiento.
- Un objeto debe poder ser capaz de cambiar su comportamiento posiblemente en tiempo de ejecución.

- Los datos que usa un algoritmo para definir un comportamiento no deben ser expuestos para algunos objetos.
- Se necesitan distintas variantes de un algoritmo.

5.1.4. Solución

La solución reside en crear jerarquías de clases que definan una familia de comportamientos para ser reutilizados por diferentes objetos que deban poseer esos comportamientos definidos. Deben existir clases abstractas, o interfaces de clase, que definan estas familias de comportamientos, y permita a los objetos mantener referencias a través de ellas a comportamientos concretos adquiridos dinámicamente.

El patrón Strategy aparece como una alternativa a la herencia, donde cada objeto deberá mantener una referencia intercambiable dentro del grupo de comportamientos concretos definido por una familia. Cada comportamiento concreto es una estrategia concreta.

Se ofrece una alternativa a las sentencias condicionales para seleccionar el comportamiento deseado al encapsular el comportamiento en clases de estrategias concretas separadas.

Permite una elección entre diferentes implementaciones de un mismo comportamiento, donde se puede optar por distintas estrategias dependiendo, por ejemplo, del tiempo de procesador y el espacio en memoria que se disponga.

5.1.5. Estructura

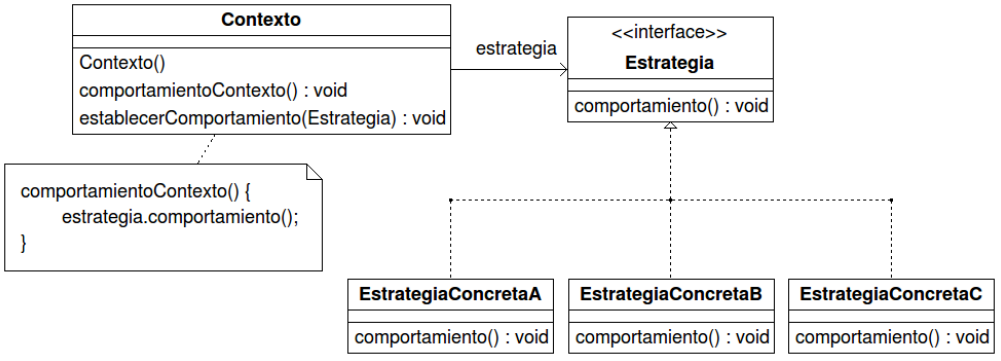

FIGURA 5.1: Estructura de clases del patrón Strategy.

- Estrategia
  Define una interfaz común a una familia de comportamientos concretos permitidos. `Contexto` mantiene una referencia a una estrategia concreta a través de la interfaz `Estrategia` para llamar al comportamiento encapsulado.

- EstrategiaConcreta
  Implementa la interfaz `Estrategia` y por lo tanto un comportamiento específico. Su estado puede permanecer inaccesible para el contexto.

- Contexto
  Se inicializa y configura con una `EstrategiaConcreta`, y acepta cambios de comportamientos en tiempo de ejecución. De ser necesario, `Contexto` puede pasarse a sí mismo, a través de una interfaz, a una estrategia concreta para que esta acceda a sus datos. `Contexto` mantiene una referencia a un objeto del tipo `Estrategia`.

5.1.6. Dinámica

El objeto `Contexto` redirige o delega las peticiones de sus clientes a su estrategia y así se define su comportamiento. Los clientes pueden elegir y crear, según las circunstancias requeridas, el objeto `EstrategiaConcreta` específico para establecer y definir el comportamiento del objeto `Contexto` con el cual interactúan. Todo esto puede ser realizado en tiempo de ejecución.

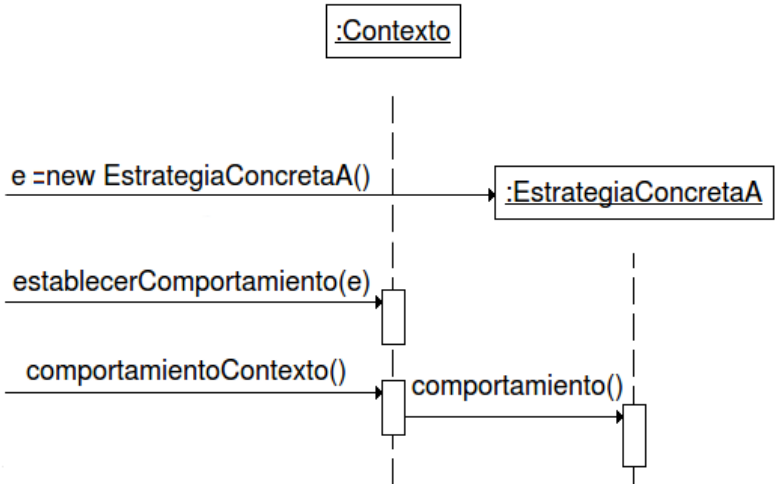

FIGURA 5.2: Estableciendo un comportamiento con el patrón Strategy.

5.1.7. Ejemplo

A continuación se mostrará un ejemplo que se verá favorecido por la aplicación del patrón Strategy. Se quiere medir el tiempo de ejecución destinado a ordenar un arreglo con diferentes métodos de ordenación. En el ejemplo solo se mostrará para dos métodos de ordenación, pero puede extenderse sin ningún problema a otros métodos.

Como primer paso debemos definir las interfaces para la familia de comportamientos o estrategias, y luego el contexto que mantendrá una referencia a una estrategia. En este caso se necesita la interfaz `MetodoOrdenacion` que define las familias que encapsulan el conocimiento de los diferentes de métodos de ordenación.

```
public interface MetodoOrdenacion {
    public void ordenar(double[] arreglo);
}
```

Código 5.1: Interfaz para las familias de métodos de ordenación.

Tanto el contexto como la estrategia deben permitir a una estrategia concreta acceder a cualquier dato que esta necesite del contexto y viceversa. La interfaz `MetodoOrdenacion` define el método `ordenar(double[] arreglo)` que permite el paso de parámetro de un arreglo de elementos a ser ordenados por alguna estrategia concreta. El contexto en este ejemplo quedará determinado por la clase `ServicioOrdenacion` que delega en la estrategia el trabajo de ordenar los elementos del arreglo.

```
public class ServicioOrdenacion {
    private MetodoOrdenacion metodo = null;
    public void ordenar(double[] arreglo) {
        metodo.ordenar(arreglo);
    }
    public MetodoOrdenacion obtenerMetodoOrdenacion() {
        return metodo;
    }
    public void establecerMetodo(MetodoOrdenacion metodo) {
        this.metodo = metodo;
    }
}
```

Código 5.2: Contexto de uso de las estrategias de ordenacón.

Otra posibilidad sería que un contexto se pase a si mismo como argumento, y que la estrategia explícitamente extraiga los datos del contexto.También es posible que la estrategia guarde una referencia a su contexto en el momento de su creación, para eliminar la necesidad de pasar argumentos en las llamadas. Pero en este caso, el `ServicioOrdenacion`

como contexto es tan simple que esto no sería necesario, además estas alternativas provocarían un acople más fuerte entre el contexto y la estrategia. Las necesidades de la estrategia concreta, sus requisitos de datos, y conocimiento del contexto determinarían qué alternativa sería mejor implementar.

Luego es necesario definir las estrategias concretas, que en este caso corresponden a los métodos de ordenación específicos que implementan la interfaz `MetodoOrdenacion`.

```
public class BubbleSort implements MetodoOrdenacion {
    public void ordenar(double[] arreglo) {
        for (int i = 0; i < arreglo.length; i++) {
            for (int j = 1; j < arreglo.length - i; j++) {
                if (arreglo[j - 1] > arreglo[j]) {
                    double aux = arreglo[j - 1];
                    arreglo[j - 1] = arreglo[j];
                    arreglo[j] = aux;
                }
            }
        }
    }
}
```

Código 5.3: Estrategia concreta de ordenación.

Cualquier otro método de ordenación que queramos implementar para medir su tiempo de ejecución, es necesario que implemente la interfaz `MetodoOrdenacion`.

```
public class QuickSort implements MetodoOrdenacion {
    ...
}
```

Código 5.4: Otra estrategia concreta de ordenación.

Sólo falta crear el cliente que use la clase `ServicioOrdenacion` para medir los tiempos de ejecución de los diferentes métodos de ordenación.

```
public class ClienteOrdenacion {
    public static void main(String[] args) {
    int n = 50000;
        double[] arregloOriginal = Datos.obtenerDatosDesordenados(n);
        double[] arreglo;
        ServicioOrdenacion contexto = new ServicioOrdenacion();
        Cronometro tiempo = new Cronometro();

        contexto.establecerMetodo(new QuickSort());
        arreglo = arregloOriginal.clone();
        tiempo.comenzar();
        contexto.ordenar(arreglo); //Comienza a ordenar QuickSort
```

```
        tiempo.detener();
        System.out.println("QuickSort: " + tiempo.tiempoTranscurrido());

        contexto.establecerMetodo(new BubbleSort());
        arreglo = arregloOriginal.clone();
        tiempo.comenzar();
        contexto.ordenar(arreglo); //Comienza a ordenar BubbleSort
        tiempo.detener();
        System.out.println("BubbleSort:" + tiempo.tiempoTranscurrido());
    }
}
```

Código 5.5: Cliente ordenando un arreglo.

Para medir el tiempo necesitaremos una clase `Cronometro` que toma el tiempo a las distintas ejecuciones de los métodos de ordenación, pero esta clase no es relevante en el patrón Strategy.

Se puede ver que el cliente establece el comportamiento del objeto contexto en tiempo de ejecución para poder medir dos métodos de ordenación con diferente implementación de algoritmo de ordenación en un mismo objeto.

5.1.8. Principios

En el código de ejemplo se puede notar que en el programa principal existe una ortogonalidad o uniformidad en la forma en cómo se cronometran los tiempos de ejecución de los diferentes algoritmos para ordenar un arreglo, sólo es necesario establecer un nuevo comportamiento al contexto, y luego enviar el mensaje ordenar. Las clases concretas encargadas de ordenar encapsulan la parte que varía, o sea, los diferentes algoritmos de ordenación. Con esto se logra cerrar las clases que representan los algoritmos de ordenación, y cada vez que se necesite extender la aplicación para que pueda cronometrar nuevos algoritmos sólo es necesario extender de la interfaz `MetodoOrdenacion` y encapsular el nuevo algoritmo. En otras palabras se está diciendo que se aplicó el principio de diseño Open-Closed (Sección 3.2.5).

La dependencia o acople entre el contexto `ServicioOrdenacion` y las subclases concretas, que implementan los algoritmos de ordenación, se reduce al utilizar la interfaz `MetodoOrdenacion` para hacer referencia a ellas. En definitiva, dentro de la clase `ServicioOrdenacion` se está programando para una interfaz, no para una implementación concreta (Sección 3.2.1) ya que nunca se hace referencia a tipos concretos.

Para un cliente, la clase `ServicioOrdenacion` ordena cuando recibe el mensaje `ordenar()`, pero en realidad delega esta responsabilidad en las clases concretas de ordenación. La

clase `ServicioOrdenacion` no es un método de ordenación[1], sino que tiene un método de ordenación[2]. Se está favoreciendo la composición de objetos sobre la herencia de clases (Sección 3.2.2).

5.2. Patrón Factory Method

Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.[GHJV95]

5.2.1. Otros nombres

Este patrón también es conocido como *Virtual Constructor*.

5.2.2. Contexto

Creación de objetos apropiados en el momento apropiado.

5.2.3. Problema

La instanciación de objetos es una actividad que no debe hacerse en cualquier parte de un sistema, deben existir clases que carguen con esta responsabilidad. Cada vez que instanciamos un objeto estamos dependiendo de la implementación de una clase concreta, no de una interfaz, lo que resulta en un código frágil y menos flexible.

El problema surge cuando queremos disponer de una estructura organizativa y funcionalidades bases para crear una aplicación, como lo sería disponer de un framework[3]. El fin de un framework es el de brindar estas funcionalidades bases como lo haría cualquier librería de clases, o tal vez una API[4], pero además define una estructura que organiza la aplicación, a la cual debemos adherirnos para aprovechar los mecanismos internos que nos brinda.

Los frameworks deben usar clases abstractas para definir y mantener relaciones entre objetos. Muchos de éstos objetos serán instancias de clases nuevas, inexistentes hasta

[1] Relación "IS-A".
[2] Relación "HAS-A".
[3] Un framework representa un marco de trabajo para facilitar el desarrollo de software.
[4] API "Application Programming Interface" o Interfaz de Programación de Aplicaciones es un software completo que nos brinda interfaces que nos permiten aprovechar sus funcionalidades.

el momento en que sean definidas por el desarrollador que haga uso del framework. Los mecanismos internos del framework también son muchas veces responsables de la creación de estos objetos, en ellos está el conocimiento de cuándo crear nuevos objetos, pero no está el conocimiento de qué tipos concretos de objetos crear. No se puede predecir qué subclases definirá el desarrollador que use el framework. Entonces, en un framework existe el dilema de instanciar objetos de clases concretas cuando sólo se conocen las clases abstractas o interfaces, las cuales no pueden ser instanciadas.

5.2.4. Solución

El conocimiento de qué clases concretas instanciar debe residir, sin duda, fuera de los mecanismo del framework, que solamente conoce el momento adecuado de instanciación de estos objetos. El patrón Factory Method encapsula el conocimiento acerca de qué subclase instanciar en un método abstracto de fabricación, y deja a disposición del desarrollador, que aprovecha los mecanismos del framework, su implementación que incluirá la creación del objeto apropiado a instanciar.

Factory Method define una interfaz para el objeto a crear, pero deja la elección del tipo concreto a las subclases, de este modo difiere su instanciación al momento de ejecución. Entonces, la clase del framework responsable de crear el objeto se apoya en sus subclases que implementan el método de fabricación, de manera que este método devuelva una instancia de la subclase apropiada en el momento apropiado.

5.2.5. Estructura

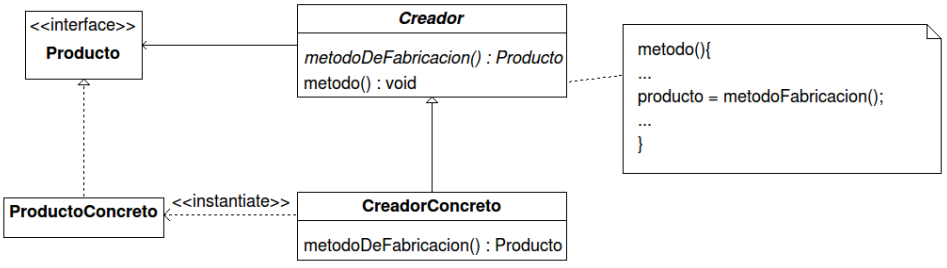

FIGURA 5.3: Estructura de clases del patrón Factory Method.

- Producto
  Define la interfaz que implementan los objetos creados a través de `metodoDeFabricacion()`.
- ProductoContreto
  Puede ser cualquier clase concreta que implemente la interfaz `Producto`. Si éstas

clases comparten una lógica en común, entonces `Producto` puede ser una clase abstracta en vez de una interfaz.

- Creador
  Clase abstracta que declara `metodoDeFabricacion()` como abstracto a ser implementado por sus subclases. Este método podría ser concreto y definir una implementación predeterminada para que devuelva un objeto concreto por defecto del tipo `Producto`. `Creador` contiene el conocimiento de cuándo crear un producto en `metodo()`, donde llama a `metodoDeFabricacion()` para crear el producto concreto apropiado.

- CreadorConcreto
  Implementa o redefine `metodoDeFabricacion()` para devolver el objeto apropiado de una clase concreta de producto.

Si estamos definiendo un framework, la clase `Creador` y la interfaz `Producto` serían parte del mismo, y brindarían el mecanismo, que puede ser tan complejo como se desee, para crear productos concretos en el momento apropiado.

5.2.6. Dinámica

En `metodo()` reside el conocimiento de cuándo crear un objeto del tipo producto, pero no qué tipo de producto concreto crear. El desarrollador del framework extiende de la clase `Creador` para añadir su propia clase `CreadorConcreto`, donde implementará `metodoDeFabricacion()` para agregar el conocimiento de qué producto concreto debe crearse. El llamado a `metodo()` deriva en un llamado a `metodoDeFabricacion()` de la clase `CreadorConcreto`, de esta forma el mecanismo puede decidir en tiempo de ejecución el objeto apropiado a crear.

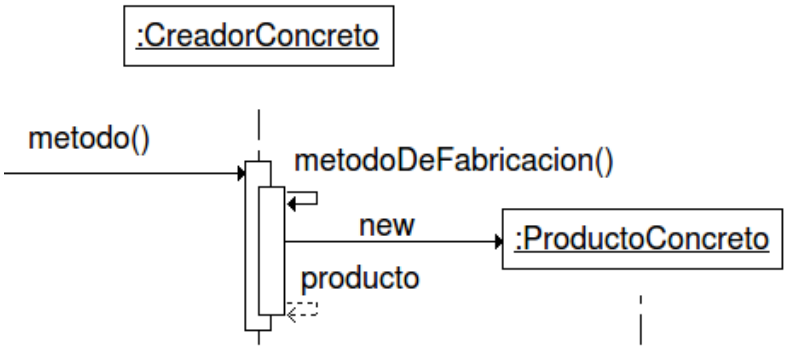

FIGURA 5.4: Creación de un producto concreto con el patrón Factory Method.

5.2.7. Ejemplo

La necesidad de implementar el patrón Factory Method es muy común, por ejemplo, cuando no se puede anticipar el tipo de objetos a crear. Este patrón es considerado como un patrón de implementación o de codificación porque está directamente relacionado con la programación, como se dijo en la Sección 4.5.

Al ser un patrón de implementación, mostrar un ejemplo resulta básicamente en mostrar su estructura, en especial en el patrón *Factory Method* por su simpleza. Por esta razón se posterga su ejemplo para cuando se muestre la aplicación del patrón MVC.

5.2.8. Principios

Siempre es más flexible crear objetos dentro de una clase con un método de fabricación que hacerlo directamente en cualquier lugar del código. En el patrón *Factory Method*, como en todos los patrones de creación, se propone encapsular lo concerniente a la creación de objetos y presentar una interfaz para tal motivo. Con esto se logra que el código que necesita instanciar objetos no varíe si en algún momento es necesario cambiar los tipos concretos a instanciar. De este modo estamos cerrando para la modificación las clases que hacen uso de éstos objetos concretos haciendo uso del principio de diseño Apertura-Clausura visto en Sección 3.2.5.

Resulta notorio el uso de interfaces o clases abstractas al momento de programar un framework, en vez de programar para una implementación concreta. El uso del principio de programar para una interfaz, no para una implementación (Sección 3.2.1) es la única opción al momento de crear un framework.

5.3. Patrón Observer

El propósito del patrón de diseño Observer[5] es el de ayudar a mantener sincronizado el estado de objetos cooperantes en un sistema. Para alcanzar esto define un mecanismo de propagación de cambios en un solo sentido, donde un solo objeto contiene el estado, y ante cualquier cambio en el mismo se notifican y actualizan automáticamente todos los objetos dependientes.

5.3.1. Otros nombres

Este patrón también es conocido como *Publisher-suscriber*, o *Dependents*.

[5]Por su amplio uso se mantendrá el término en inglés en vez de Observador.

5.3.2. Contexto

Objetos relacionados con estados sincronizados.

5.3.3. Problema

El problema surge cuando los datos que mantiene un objeto cambian y muchos objetos dependen de estos datos. El ejemplo clásico que se usa para representar este problema es el de las interfaces gráficas de usuarios, donde los datos son representados visualmente de diferentes formas por diferentes vistas. Cuando existe un cambio en un dato, todas las vistas que dependen de este dato deben actualizarse para representar consistentemente esta información. Se podría llamar directamente a las vistas desde el objeto que tiene los datos y propagar así la información a los objetos dependiente, pero no sería una solución flexible ni reutilizable.

Restricciones y requerimientos que debe tener la solución:

- Uno o más objetos deben ser notificados acerca del cambio de estado en un objeto en particular.
- El número de objetos dependientes no se conoce a priori, y puede cambiar en tiempo de ejecución.
- Las clases que definen los datos y los objetos dependientes deben poder reutilizarse en forma independiente.
- No es factible que los objetos dependientes extraigan directamente información sin ser notificados de un cambio.
- El mecanismo de propagación de información no debe crear un fuerte acoplamiento entre los objetos dependientes y el objeto que posee los datos.

5.3.4. Solución

El patrón Observer describe cómo balancear las fuerzas que configuran los requerimientos y restricciones para obtener una solución reutilizable y flexible. Los objetos claves en este patrón son: el **sujeto**, que posee los datos, y el **observador**, que es el objeto dependiente que debe sincronizar su estado según los datos del sujeto.

Un sujeto puede tener un número cualquiera de observadores, y cada vez que el sujeto cambie su estado los observadores serán notificados para que sincronicen su propio

estado con el estado del sujeto. Este tipo de interacción también es conocida como Publicador-Suscriptor.[6] El sujeto es quien publica los cambios, y los objetos suscriptos u observadores, los que serán notificados. El sujeto envía notificaciones sin conocer realmente a sus observadores suscriptos.

El sujeto mantiene un registro de los observadores suscriptos. Un objeto puede suscribirse como observadoren cualquier momento, como así también cancelar la suscripción y dejar de recibir notificaciones.

El sujeto decide qué cambio de estado interno debe ser notificado a los observadores, pudiendo dejar de lado algunos cambios que no sean necesario notificar. También puede encolar diferentes cambios antes de realizar una notificación.

Un objeto puede ser observador de diferentes sujetos y al mismo tiempo puede ser sujeto de diferentes observadores. Puede desempeñar los dos roles a la vez, tanto de observador como de publicador.

5.3.5. Estructura

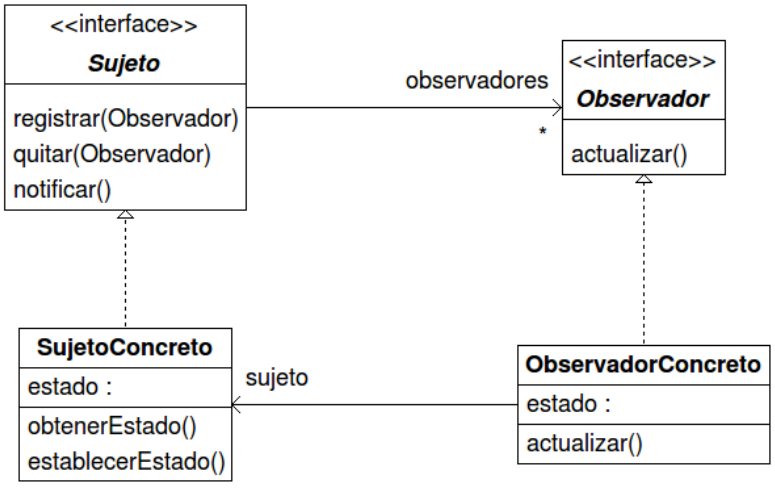

FIGURA 5.5: Estructura de clases del patrón Observer.

- Sujeto

  Mantiene una lista de observadores, y los conoce sólo a través de la interfaz `Observador`. No conoce las clases concretas de ningún observador, el sujeto solamente conoce lo que le interesa saber.

  Proporciona una interfaz para agregar y quitar objetos del tipo Observador.

[6] Del inglés "Publisher-Suscriber", a veces traducido Publicar-Suscribir.

- Observador
  Interfaz para actualizar los objetos que deben ser notificados ante cambios en el estado del sujeto. Establece el estilo de actualización, *Pull* o *Push*, si el método actualizar observadores especifica o no parámetros en su firma.

- SujetoConcreto
  Mantiene el estado que es de interés para los objetos observadores. Es responsable en notificar a todos sus observadores cuando su estado cambia.

- ObservadorConcreto
  Implementa la interfaz `Observador`, que es la cara que muestra a los sujetos concretos. Posee una referencia al objeto `SujetoConcreto` que observa. Mantiene su propio estado sincronizado con el estado del sujeto

5.3.6. Dinámica

Diferentes implementaciones de este patrón suelen hacer que la cantidad de información que envía un sujeto varíe. De acuerdo a esto se han definido dos modelos en el patrón Observer. En un extremo está el modelo *Push*, donde el sujeto envía en cada notificación información detallada acerca del cambio a los observadores, la necesiten o no. En el modelo *Pull*, el sujeto sólo envía la notificación mínima de que ocurrió un cambio, y es tarea de los observadores traer la información que necesiten. Muchas variaciones son posibles entre estos dos modelos. El modelo *Push* tiene un comportamiento dinámico muy rígido en su comunicación, comparado con el modelo *Pull* que es más flexible pero a la vez produce una mayor cantidad de mensajes entre sujetos y observadores.

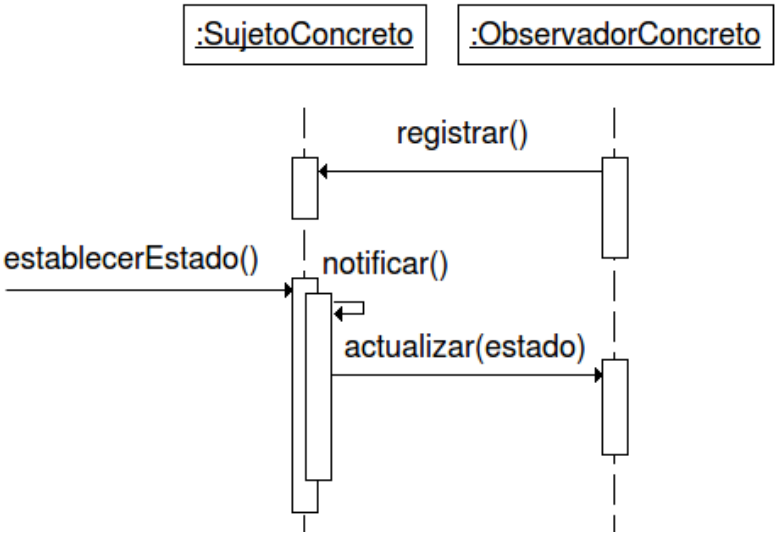

FIGURA 5.6: Modelo *Push* del patrón Observer.

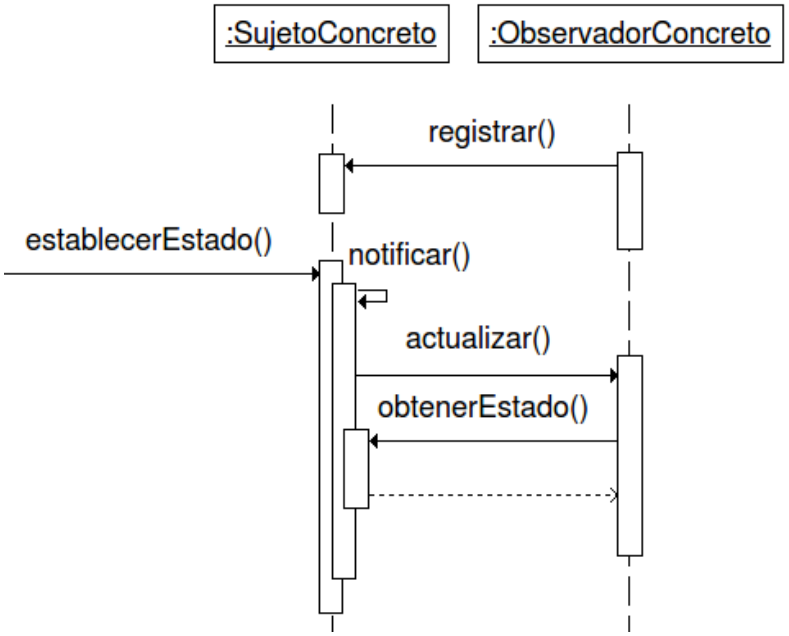

FIGURA 5.7: Modelo *Pull* del patrón Observer.

5.3.7. Ejemplo

Supongamos una estación climatológica que posee sensores de temperatura, humedad, y presión atmosférica. Los sensores realizan las mediciones del clima y las transforman en datos que son enviados a la estación, la cual es responsable de mantener estos datos. Cuando algunos de los sensores detecta un cambio en el clima, se envía el mensaje `setMedidas` con los datos censados a la estación climatológica.

```
public class EstacionClima {
    private float temperatura;
    private float humedad;
    private float presion;
    private  VisorCondicionesActuales visor;
    ...
    public void setMedidas(float temperatura, float humedad, float presion) {
        this.temperatura = temperatura;
        this.humedad = humedad;
        this.presion = presion;
        visor.setCondiciones(temperatura, humedad);
        visor.mostrar();
    }

    public float getTemperatura() {
        return temperatura;
    }

    public float getHumedad() {
```

```
        return humedad;
    }

    public float getPresion() {
        return presion;
    }
}
```

Código 5.6: Estación Climatológica representada en una clase.

También se dispone de un visor de las condiciones actuales, que muestra por consola de texto la temperatura y humedad cuando se registra un cambio en las medidas.

```
public class VisorCondicionesActuales{
    private float temperatura;
    private float humedad;

    public VisorCondicionesActuales(float temperatura, float humedad) {
        setCondiciones(temperatura, humedad);
    }
    public void mostrar() {
        System.out.println("Condiciones actuales: " + temperatura +
        "C  " + humedad + "% humedad");
    }
    public void setCondiciones(float temperatura, float humedad) {
        this.temperatura = temperatura;
        this.humedad = humedad;
    }
}
```

Código 5.7: Visor de condiciones actuales.

Luego de un tiempo, la estación climatológica requiere de dos nuevos visores, uno para mostrar el pronóstico del tiempo y otro para mostrar unas estadísticas sobre el clima en base a datos anteriores. Sin duda debemos cambiar el diseño existente de nuestra aplicación si queremos que sea flexible y reutilizable.

Tenemos un objeto `estacionClima` que mantiene los datos de temperatura, humedad, y presión, y objetos visores que dependen de estos datos para mostrar información por consola. Llamar directamente a los visores, como se viene haciendo, no es una solución flexible ni reutilizable, y crea un fuerte acople entre los objetos que interactúan. Además es muy probable que la estación necesite de nuevos visores en un futuro próximo.

Como conocemos el patrón Observer, todo indica que estamos en un contexto donde su aplicación resultaría en el diseño flexible y reutilizable que estamos buscando.

Para lograr un acoplamiento débil entre el objeto `EstacionClima` que mantiene los datos y el objeto `VisorCondicionesActuales` que debe sincronizar su estado con estos datos,

necesitamos de las siguientes interfaces que nos permitirán exponer lo justo y necesario que cada objeto necesita conocer del otro:

```
public interface Sujeto {
    public void registrar(Observador o);
    public void quitar(Observador o);
    public void notificar();
}
```

Código 5.8: Interfaz Sujeto.

```
public interface Observador {
    public void actualizar();
}
```

Código 5.9: Interfaz Observador.

Con la interfaz `Observador` así definida, la información que el sujeto transmita será solamente la notificación a través del mensaje `actualizar()`, sin transferir ningún parámetro. De esta forma estamos siguiendo el estilo de actualización *Pull*.

El sujeto es el objeto que deberá mantener los datos, por lo tanto, en este caso, la clase `EstacionClima` será un sujeto concreto que deberá implementar la interfaz `Sujeto` y mantener una lista de todos los observadores registrados.

Cuando los sensores envíen nuevas medidas climatológicas, debemos notificar a los observadores registrados para que actualicen su estado. Al momento de la notificación podemos verificar si los datos son confiables, ya que los sensores pueden enviar datos erróneos o ser muy sensibles y las lecturas podrían fluctuar entre variaciones mínimas. Así podemos optimizar el mecanismo de notificación y decidir si es necesario notificar de los cambios a los observadores.

```
public class EstacionClima implements Sujeto {
    private ArrayList<Observador> observadores = new ArrayList<Observador>();
    private float temperatura;
    private float humedad;
    private float presion;

    public void registrar(Observador o) {
        observadores.add(o);
    }

    public void quitar(Observador o) {
        observadores.remove(o);
    }
```

```
    public void notificar() {
        for (Observador o : observadores) {
            o.actualizar();
        }
    }
    private void medidasCambiadas() {
        //decidir si es necesario notificar
        notificar();
    }
    public void setMedidas(float temperatura, float humedad, float presion) {
        this.temperatura = temperatura;
        this.humedad = humedad;
        this.presion = presion;
        medidasCambiadas();
    }
    //getters
}
```

Código 5.10: Estación Climatológica con el rol de Sujeto.

Los visores tendrán el rol de observador. Cada visor concreto implementará la interfaz `Observardor`, y a la vez mantendrá una referencia al objeto sujeto `estacionClima` para realizar su actualización siguiendo el modelo *Pull*, donde cada observador es responsable de adquirir los datos que les son de utilidad. Como los observadores concretos en este caso también serán visores, implementaremos la interfaz `Visor`, que solamente incluirá el método `mostrar()` que será llamado cuando los elementos o datos de los visores tengan que ser mostrados por consola.

```
public class VisorCondicionesActuales implements Observador, Visor {
    private float temperatura;
    private float humedad;
    private EstacionClima estacionClima;

    public VisorCondicionesActuales(EstacionClima weatherData) {
        this.estacionClima = weatherData;
        this.estacionClima.registrar(this);
    }
    public void mostrar() {
        System.out.println("Condiciones actuales: " + temperatura
                + "C  " + humedad + "% humedad");
    }
    public void actualizar() {
        this.temperatura = estacionClima.getTemperatura();
        this.humedad = estacionClima.getHumedad();
        mostrar();
    }
}
```

Código 5.11: Visor de condiciones actuales con el rol de Observador y de Visor.

La suscripción del observador ante el sujeto sucede en el momento de la creación del visor, esto puede estar implementado de diferentes maneras, el patrón Observer no establece dónde debe hacerse. Debemos hacerlo donde sea conveniente para nuestro proyecto en particular.

Para el visores que mostrará un pronóstico y el visor de estadísticas meteorológicas, sólo debemos crear sus respectivas clases e implementar las interfaces de `Observador` y `Visor`. Cada una de éstas clases mantendrá datos que les serán útiles para cumplir con sus responsabilidades, y al adherirse a la interfaz de `Observador` tienen la posibilidad de registrarse con el objeto `estacionClima` y mantener sincronizados sus estados.

5.3.8. Principios

El principio de diseño fundamental en el que el patrón Observer se basa es el de acoplamiento débil (Sección 3.2.4). El patrón Observer propone la interacción entre objetos sujetos y objetos observadores sobre un mínimo conocimiento entre ellos, el necesario y justo para lograr la comunicación y mantener la coherencia entre los estados de los diferentes objetos.

Lo único que un sujeto conoce de un observador es que implementa una cierta interfaz, no necesita conocer el tipo concreto del observador.

Este acoplamiento débil permite registrar y quitar observadores en cualquier momento y hasta en tiempo de ejecución, porque en lo único que depende el sujeto para esto es en la colección de objetos que implementan la interfaz observador.

No necesitamos modificar el sujeto concreto para agregar nuevos tipos como observadores, solo deben implementar la interfaz Observador. La clave en lograr el desacople está en el uso de interfaces para minimizar la interdependencia en los objetos que interactúan. Se está programando para una interfaz, no para una implementación (Sección 3.2.1).

5.4. Patrón Composite

El patrón Composite nos permite componer objetos dentro de estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los objetos compuestos.

5.4.1. Contexto

Aplicación de operaciones a colecciones de objetos.

5.4.2. Problema

Cuando debemos manejar colecciones de objetos, en las que puede encontrarse objetos que son colecciones por si mismos, recorrer cada elemento de la colección y tratar las subcolecciones por separados resulta en un código complejo.

Una colección que pueda llegar a contener en forma anidada otra colección determina una estructura de árbol. Discriminar una hoja de un nodo en una estructura de árbol para aplicar una cierta operación sobre los elementos que componen el árbol resulta en un código tendencioso al error.

Trabajar con distintos elementos contenedores, o elementos que son colecciones, puede resultar complejo, ya que pueden estar implementados de distintas formas. Recorrerlos para aplicar una operación a los elementos que los componen sería posible solamente si conocemos sus implementaciones. Además, cada vez que se agregue un elemento nuevo deberíamos modificar el código para tratar esos objetos según su implementación.

La solución debe cumplir con las siguientes restricciones y requerimientos:

- Tratar sin diferencias tanto nodos como hojas sobre operaciones definidas.
- Iterar fácilmente sobre todos los elementos del árbol.
- Facilitar la adición de nuevos tipos de componentes a la estructura de árbol.
- Liberar al cliente de la colección sobre detalles de implementación de los distintos elementos contenedores de la colección para iterar sobre ellos.

5.4.3. Solución

La clave de la solución está en permitir manipular de manera uniforme tanto las hojas o elementos individuales como los nodos o elementos contenedores. Para esto se define una interfaz común con las operaciones que el cliente espera encontrar en los elementos de la estructura. Además, la interfaz deberá ofrecer operaciones para manejar los componentes de un objeto compuesto.

Se define una jerarquía de clases formada por elementos hojas y elementos compuestos. Los objetos hojas pueden ser parte de una colección de un objeto componente más

complejo, y éste a su vez componer un objeto aún más complejo. Donde en el código se espere un objeto hoja, también se podrá recibir un objeto compuesto. A los clientes no les debe importar ni conocer si están tratando con un componente hoja o un componente compuesto, esto define una ortogonalidad en el tratamiento de los elementos que simplifica el código del cliente.

Al brindar una interfaz para añadir elementos a una colección, luego si se definen nuevas subclases hojas o compuestos, éstas deberían funcionar sin problemas en el código del cliente, sin hacer ninguna intervención o modificación.

5.4.4. Estructura

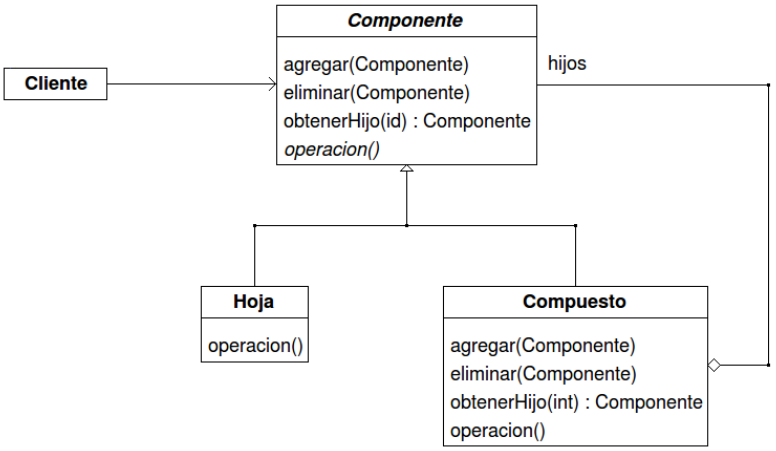

FIGURA 5.8: Diagrama de clases del patrón Composite.

- Componente
  Clase abstracta que define una interfaz para todos los objetos que forman parte de la estructura de árbol. `Componente` implementa el comportamiento predeterminado común a todos los objetos de una composición, y declara una interfaz para manejar los componentes hijos.
  Opcionalmente se puede declarar una interfaz para acceder al padre de un componente.

- Hoja
  Representa los objetos componentes individuales que no tienen hijos, y define el comportamiento de éstos objetos en la composición o estructura de árbol.

- Compuesto
  Define el comportamiento de los objetos componentes que tienen hijos. Son responsables de mantener referencias a sus hijos, e implementan la parte de la interfaz relacionada con el manejo de los hijos.

- Cliente
  Manipula objetos de la composición a través de la interfaz `Componente`.

5.4.5. Ejemplo

Supongamos que queremos crear una aplicación para dibujar figuras. Una implementación simple debería definir clases para figuras primitivas, como por ejemplo una línea. También debe ser posible agrupar distintas figuras primitivas para formar una figura más compleja. Además aplicar operaciones para transformar las figuras no debe agregar complejidad al cliente que las dibuje. Por ejemplo si queremos trasladar una figura de su posición actual a otra posición, o tal vez rotarla.

Sería un problema que el código que usa estas clases deba tratar en formas diferentes a los objetos primitivos, que representan una figura simple como una linea, de un objeto contenedor, que representa una figura más compleja. Para evitar éste problema es necesario definir una interfaz para éstos dos tipos de objetos.

```
public abstract class Figura implements Cloneable {
    public void agregar(Figura figura) {
        throw new UnsupportedOperationException();
    }
    public abstract void dibujar(Graphics g);
    public abstract Figura trasladar(int x, int y);
}
```

Código 5.12: Interfaz común para tratar objetos individuales y objetos compuestos.

La clase abstracta `Figura` representa la clase abstracta `Componente` de la estructura del patrón *Composite*. Ésta clase también podría definir alguna implementación común para sus subclases, como una operación para copiar figuras, la cual debería ser sobreescrita en la subclase que instancie objetos contenedores.

La clase que representa una línea sería la figura primitiva en ésta aplicación, con la que a partir de ella podríamos formar figuras más complejas. Ésta clase simplemente modela una línea a partir de sus puntos extremos, e implementa los métodos abstractos

`dibujar(Graphic g)`[7], y `trasladar(int x, int y)`. Como es una figura primitiva, no puede tener figuras hijas, por lo tanto no sobre-escribe la operación para agregar figuras.

```
import java.awt.Graphics;

public class Linea extends Figura {
    private int x1, y1, x2, y2;
    public Linea(int x1, int y1, int x2, int y2) {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }
    public void dibujar(Graphics g) {
        g.drawLine(x1, y1, x2, y2);
    }
    public Figura trasladar(int x, int y) {
        this.x1 += x;
        this.y1 += y;
        this.x2 += x;
        this.y2 += y;
        return this;
    }
}
```

Código 5.13: Clase que representa una figura primitiva.

La clase `PoliLinea` define una agregación de objetos del tipo figura, que pueden ser tanto figuras primitivas como figuras compuestas. El método para dibujar delega en sus figuras hijas la tarea de dibujarse a sí misma, y en el caso de ser una figura hija compuesta, ésta delegará en sus hijas la tarea de dibujarse, y así sucesivamente. `PoliLinea` representa las figuras compuestas, por lo tanto debe sobre-escribir el método `agregar(Figura figura)` para componer una figura compuesta.

```
public class PoliLinea extends Figura {
    List<Figura> lineas = new ArrayList();
    @Override
    public void agregar(Figura figura) {
        lineas.add(figura);
    }
    public void dibujar(Graphics g) {
        for (Figura s : lineas) {
            s.dibujar(g);
        }
    }
    public Figura trasladar(int x, int y) {
```

[7]La manera de dibujar una línea va a depender de la plataforma gráfica que se dispone. En éste ejemplo se usa de la plataforma Java el paquete `java.awt` "Abstract Window Toolkit" que contiene clases para crear ventanas y dibujar gráficos e imágenes.

```
        for (Figura s : lineas) {
            s.trasladar(x, y);
        }
        return this;
    }
}
```

Código 5.14: Clase que representa una figura compuesta.

El cliente manipula los objetos en la composición a través de la interfaz `Figura`. Puede dibujar y realizar cualquier operación de transformación sobre las figuras. En éste caso el cliente es el papel donde se dibujan éstas figuras, y dibujarlas resulta en iterar por cada una de ellas y llamar al método para dibujar. De la misma manera se podría aplicar cualquier operación de transformación sobre ellas, como por ejemplo trasladarlas. Gracias al patrón Composite, el cliente que manipula éstos objetos resulta sencillo de implementar.

```
import java.awt.Graphics;
import javax.swing.JFrame;

public class Papel extends JFrame {
    private Figura figura = new PoliLinea();
    public Papel() {
        super("Graficador");
        setLayout();
    }
    private void setLayout(){
        setSize(600, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }
    @Override
    public void paint(Graphics g) {
        super.paint(g);
        figura.dibujar(g);
    }
    public void dibujar(Figura figura) {
        this.figura = figura;
        this.repaint();
    }
}
```

Código 5.15: Cliente que dibuja figuras.

La clase `Papel` es una ventana gráfica porque hereda de `JFrame`[8]. Cada vez que es necesario dibujar la ventana, el cliente envía el mensaje dibujar a la figura que debe representar

[8]Se hace uso de la biblioteca gráfica Swing de la plataforma Java. `JFrame` representa una ventana y dispone del método `public void paint(Graphics g)` que es llamado para dibujar su contenido.

gráficamente. Si la figura que debe dibujar es una `Linea`, se trata directamente, y si es una `PoliLinea` se redirige las peticiones a sus componentes hijos.

Por último se muestra un código principal que dibuja un triángulo, una cruz, y una línea, para testear la aplicación para dibujar figuras que se creó.

```
public class Test {
    public static void main(String[] args) throws CloneNotSupportedException {
        Figura figura = new PoliLinea();
        Figura linea = new Linea(250, 150, 350, 150);

        Figura triangulo = new PoliLinea();
        Figura a = new Linea(150, 100, 100, 200);
        Figura b = new Linea(150, 100, 200, 200);
        Figura c = new Linea(100, 200, 200, 200);
        triangulo.agregar(a);
        triangulo.agregar(b);
        triangulo.agregar(c);

        Figura cruz = new PoliLinea();
        Figura linea1 = new Linea(100, 100, 200, 200);
        Figura linea2 = new Linea(200, 100, 100, 200);
        cruz.agregar(linea1);
        cruz.agregar(linea2);

        figura.agregar(triangulo);
        figura.agregar(linea);
        figura.agregar(cruz.trasladar(300, 0));

        Papel papel = new Papel();
        papel.dibujar(figura);
    }
}
```

Código 5.16: Dibujando figuras.

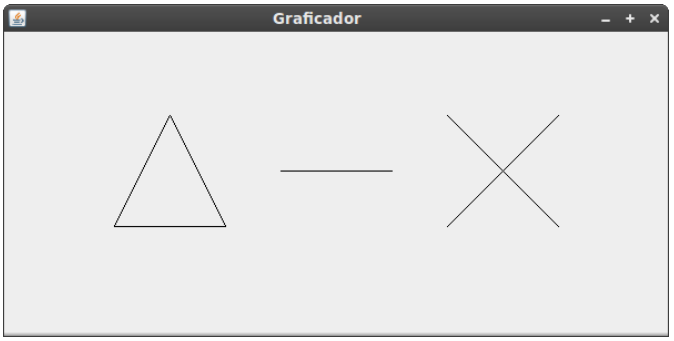

Figura 5.9: Ventana del graficador.

Es fácil agregar nuevas figuras para que sean primitivas en la aplicación, solo debemos heredar de `Figura` e implementar el método para dibujar. Podríamos implementar un

círculo, o texto para que sean figuras primitivas en la aplicación. Lo mismo si queremos extender la aplicación para permitir aplicar nuevas transformaciones a las figuras, como por ejemplo rotar, o cambiar de escala una figura, solo debemos agregar el método en la interfaz `Figura` e implementar en las figuras concretas el procedimiento geométrico que transformaría la figura.

5.4.6. Principios

La aplicación del principio de sustitución de Liskov (Sección 3.2.6) es fundamental en el patrón *Composite*. Los subtipos son siempre sustituibles por su tipo base `Figura`, y es lo que permite tratar de manera uniforme tanto objetos individuales como compuestos. Aunque la equivalencia de comportamiento entre los objetos hojas y objetos compuestos no es total, ya que no tiene sentido llamar al método `agregar(componente)` en un objeto hoja, para el cliente, o sea para la clase `Papel`, si existe esta equivalencia de comportamiento ya que el cliente no está interesado en componer objetos compuestos.

En todo momento se manipula los objetos a través de la interfaz `Figura`, lo que mantiene las implementaciones desacopladas y las dependencias al mínimo. Se está programando para una interfaz, no para una implementación (Sección 3.2.1).

Capítulo 6

Modelo Vista Controlador

El patrón MVC desarrollado por Trygve Reenskaug en 1979, años antes de que el concepto de patrón de diseño se instale en el área del software, cuando trabajaba como científico invitado con el grupo que desarrollaba Smalltalk en Xerox PARC.

La primera GUI con múltiples ventanas que se superponen fue originalmente diseñada para Smalltalk-80, más tarde Macintosh and Windows imitaron esta manera de mostrar las interfaces. El patrón fue mostrando muchas facetas desde su origen, a través del tiempo, de las diferentes tecnologías, y los distintos requerimientos que se exigen para una solución de software.

Este patrón representa la experiencia ganada a través de años de desarrollo de sistemas interactivos. Muchas aplicaciones conocidas en la actualidad usan el patrón MVC, es la clásica arquitectura de las aplicaciones en Smalltalk, y está presente en muchos framework de desarrollo web.

6.1. Patrón compuesto y combinación de patrones

En el diseño de un sistema de información es común que coexistan la aplicación de varios patrones, y que algunos de ellos trabajen en conjunto para alcanzar alguna meta. Un patrón también puede ser aplicado varias veces en diferentes partes de un diseño.

Hay que recordar que un patrón sólo existe en un contexto definido y común en el desarrollo de sistemas, como lo es el contexto de las aplicaciones interactivas, en donde aparecen restricciones y problemas recurrentes. Las soluciones que fueron exitosas van quedando en la experiencia colectiva y luego las descubrimos como un patrón.

Existen infinitas combinaciones de patrones que no responden a un problema común, sino que son específicas en cada situación de diseño. Son situaciones nuevas que existen en un contexto atípico. Estos tipos de combinaciones no representan un patrón compuesto. Un patrón compuesto es una combinación de patrones, pero no toda combinación de patrones es un patrón compuesto.

Un patrón compuesto es un conjunto de unos pocos patrones de diseño que están combinados para resolver un problema general común y recurrente. Así en el patrón compuesto MVC se hallan otros patrones que han sido usados una y otra vez en conjunto para resolver los problemas que surgen en las aplicaciones interactivas.

No existe una regla en las cantidad de patrones que pueden participar en un patrón compuesto, lo que importa es la intención y propósito de los elementos que lo componen para lograr la solución. Esta cantidad de patrones participantes estará determinada según las necesidades del diseño. Por lo general en una arquitectura MVC clásica se puede encontrar el patrón Observer, Strategy, Method Factory, y Composite.

6.2. Patrón MVC

El propósito fundamental del patrón Modelo Vista Controlador es el de acortar las diferencias entre el modelo mental del usuario humano y el modelo digital que existe en la computadora. La solución ideal MVC soporta la ilusión del usuario de estar viendo y manipulando directamente la información del dominio. La estructura es útil si el usuario necesita ver simultáneamente los mismos elementos del modelo en contextos diferentes y/o desde diferentes puntos de vistas. [Ree]

6.2.1. Contexto

Aplicaciones interactivas con múltiples interfaces de usuario sincronizadas.

6.2.2. Problema

Generalmente los sistemas buscan información de alguna base de datos y la presentan a los usuarios a través de una interfaz gráfica, donde el usuario también puede modificar dicha información y el sistema se encarga de salvar los cambios nuevamente en la base de datos. Como el flujo de información se da entre la base de datos y la interfaz de usuario, naturalmente se tiende a unir el mecanismo de acceso a datos con el de la interfaz de usuario. Esto disminuye la cantidad de código y aumenta la performance de la aplicación, pero este enfoque natural y simplista puede traernos grandes problemas a futuro.

Las interfaces de usuario son, por lo general, mucho más propensas al cambio que los mecanismo de acceso a datos. Por ejemplo, cuando extendemos o agregamos una funcionalidad a la aplicación, de alguna manera debemos modificar la interfaz para tener acceso a las nuevas funcionalidades. Además, al acoplar el mecanismo de acceso a datos con el código de las interfaces de usuario, se tiende luego a incorporar lógica de negocio en el mismo lugar. Esto resulta en una pésima decisión ya que va en contra de la reutilización del software (Sección 1.2.2).

Un usuario puede requerir alguna adaptación o requerimiento especial de las interfaces según su rol en el uso de la aplicación. Por ejemplo, un empleado se puede ocupar de cargar datos y requerir una mayor interacción con el teclado, mientras un gerente debe tener otra visión del sistema, más general y abarcativa, y tal vez desee interactuar con el sistema a través de iconos con el uso del mouse.

Un cambio de versión al sistema de ventanas puede implicar cambios en el código de las interfaces. Por lo general se busca implementar las interfaces con un sistema de presentación independiente de la plataforma, aunque a veces es imposible.

Construir un sistema flexible que cumpla con estos requerimientos puede llegar a ser muy costoso y propenso a errores si las interfaces se hallan acopladas con la lógica de negocio y el mecanismo de acceso a datos. Puede resultar en mantener algo casi parecido a diferentes sistemas por cada interfaz de usuario existente en la aplicación.

Las siguientes fuerzas que resultan de los requerimientos y restricciones que debe poseer una aplicación nos ayudan a entender el problema de las aplicaciones interactivas:

- La misma información debe ser presentada de diferentes formas en múltiples interfaces de usuario.
  Todas las interfaces deben estar sincronizadas con el estado de la aplicación.
- La información que se muestra en las interfaces y el comportamiento de la aplicación deben reflejar inmediatamente los cambios en los datos.

- Debe ser fácil tanto modificar como agregar nuevas interfaces de usuario, y debe ser posible hacerlo en tiempo de ejecución.

- Un sistema se debe poder portar o mover a otra plataforma con diferentes mecanismos de presentación de interfaces, sin modificar la lógica de negocio desarrollada en la aplicación. El código de presentación de las interfaces de usuario tiende a ser más dependiente de las plataformas que el de la lógica de negocio. Por ejemplo, si se quiere migrar una aplicación a un dispositivo portátil PDA (Personal Digital Assistants), teléfono celular, o a un sistema web, se deberá reemplazar gran parte del código de las interfaces, pero el código de la lógica de negocio deberá quedar intacto.

- Debe poder permitir el desarrollo en paralelo. El diseño visual de las interfaces requiere diferentes habilidades que desarrollar una lógica de negocio. Es muy raro que una persona disponga de ambas habilidades, por esto es deseable que el desarrollo de una aplicación pueda ser separada.

- Hacer más sencilla la testeabilidad del sistema. Crear tests automatizados para las interfaces de usuario es generalmente más difícil y trabajoso que realizarlo para la lógica de negocio. Reducir el acople de diferentes aspectos en el código de las interfaces de usuario mejora y facilita la testeabilidad de una aplicación.

6.2.3. Solución

El patrón arquitectónico Modelo Vista Controlador surge de la separación de los diferentes asuntos involucrados en tratar requerimientos no funcionales en el dominio de las aplicaciones interactivas.

En la interacción hombre computador están involucrados los mecanismos de entrada y salida de una aplicación, los cuales son técnicamente muy diferentes y corresponden a distintos aspectos en un sistema. Por esto el patrón MVC divide una aplicación interactiva en tres partes. El **modelo**, que contiene la funcionalidad o lógica de negocio y los datos del sistema, se halla separado y es independiente de los mecanismos de interacción. La **vista**, que es la abstracción responsable de las salidas y presentación del sistema, y el **controlador**, que se encarga de tomar e interpretar las entradas del usuario.

Los objetos del **modelo** representan la información y el comportamiento de las cuestiones funcionales acerca del dominio del problema, están despojados de lo concerniente al aspecto visual y al comportamiento de las interfaces. Contienen todos los datos y funcionalidades que serán presentados en las interfaces de usuario. En su forma más pura de Orientación a Objetos, el modelo es un objeto dentro del modelo del dominio.[Fow02]

La **vista** muestra información a los usuarios y representa la exposición del modelo en la interfaz de usuario. La vista trata solo cuestiones referidas a la presentación de información que extrae del modelo, y cada vista tiene generalmente asociado un controlador.

El **controlador** captura las entradas del usuario, manipula el modelo, y permiten que las vistas se actualicen apropiadamente. De esta forma, las interfaces de usuario son una combinación de vista y controlador.

Esta separación permite múltiples vistas de un mismo modelo. Si un usuario realiza un cambio en el estado del modelo a través del controlador de una vista, todas las vistas dependientes del dato modificado deben reflejar el cambio. O sea, existe un mecanismo de propagación de cambios, donde las vistas son notificadas por el modelo cuando subyace una modificación. En otras palabras, se hace uso del patrón Observer para mantener las vistas actualizadas.

6.2.4. Estructura

El modelo contiene el conocimiento, encapsula los datos y procesos específicos del dominio de una aplicación. Los controladores, en representación del usuario hacen uso de los métodos del modelo que representan los procesos. El modelo también provee el acceso a los datos a través de métodos que las vistas hacen uso para actualizar su presentación.

El mecanismo de propagación mantiene una colección de los objetos dependientes del estado del modelo. Todas las vistas y algunos controladores estarán registrados para ser informados por este mecanismo cuando exista un cambio en el estado del modelo.

- Modelo:
  El modelo es un conjunto de objetos que colaboran para lograr el núcleo funcional y mantener el estado de la aplicación. En una arquitectura MVC existe una parte responsable de manejar el mecanismo de propagación de cambios, que el modelo aprovecha para mantener actualizados tanto vistas como controladores.

- Vista:
  Los objetos de la vista son los encargados de mostrar al usuario la información contenida en el modelo. En un sistema pueden existir múltiples vistas que presentan esta información en diferentes formas. Cada vista implementa el método `actualizar()`, que es activado cada vez que existe un cambio en el estado del modelo, o mejor aún cuando hay un cambio en la datos de interés que serán presentados por la vista.

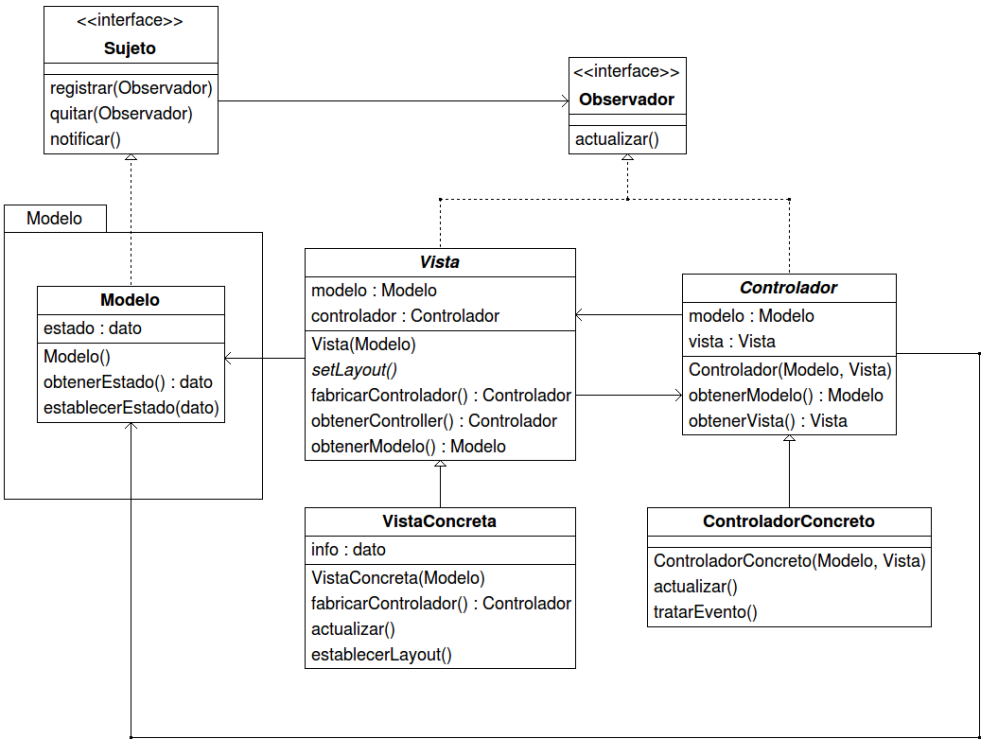

FIGURA 6.1: Estructura de clases del patrón MVC.

Durante la creación de las vistas, en su método constructor, éstas son asociadas con el modelo, y son ellas mismas las responsables de registrarse como observadores en el mecanismo de propagación de cambios. Cada vista concreta, de ser necesario, reescribirá el método `fabricarControlador()` para mantener una relación y asociarse con un controlador concreto específico. Si se decide no reescribir este método, se estará creando una vista pasiva, en la cual el usuario solo podrá observar los datos y de ninguna manera modificar e interactuar con información alguna. Se está aplicando el patrón *Method factory* como se vio en la Sección 5.2.

La relación de una vista con un controlador es más estrecha que la que tiene con el modelo. Es necesario que el controlador disponga de un conocimiento más detallado de la vista que controla. En tiempo de ejecución existe una relación uno a uno entre un objeto vista y un objeto controlador, aunque pueda haber más de una clase controlador por vista.

- Controlador:
  Los controladores estarán interesados en escuchar diferentes eventos que suceden en las vistas. La forma en cómo estos eventos suceden dependerá del sistema de ventanas que estemos usando y de la plataforma donde se encuentre la aplicación, lo cual va a influir en cómo los controladores capturen y manejen los eventos.

Cada controlador implementará métodos para reaccionar ante los eventos que son lanzados desde la vista de interés, como el método `tratarEvento()`. El controlador entonces se encargará de traducir estos eventos en peticiones al modelo, o a su misma vista asociada. De alguna manera los controladores deberán registrarse en las vistas para poder escuchar los eventos que en ellas sucedan.

Si el comportamiento de un controlador depende del estado del modelo, éste deberá registrase como observador del mismo para ser parte del mecanismo de propagación de cambios que usa el modelo para enviar notificaciones. Esto puede ser necesario, por ejemplo, cuando algún estado específico del modelo es notificado al controlador para deshabilitar algún elemento de la vista.

CUADRO 6.1: Responsabilidades de los elementos del patrón MVC

Modelo	Vista	Controlador
Implementar lógica	Componer Layout	Interpretar acciones del usuario
Notificar cambios	Observar modelo	Observar modelo
Registrar observadores	Enviar eventos al controlador	Escuchar eventos
Mantener datos	Actualizar presentación	Actualizar vista
Persistir datos	Fabricar controlador	Modificar modelo

6.2.5. Dinámica

El código que inicializa a la triada MVC usualmente se encuentra fuera del modelo, vistas, y controladores, por ejemplo en el programa principal. Es el que pone en funcionamiento los mecanismo participan en el patrón.

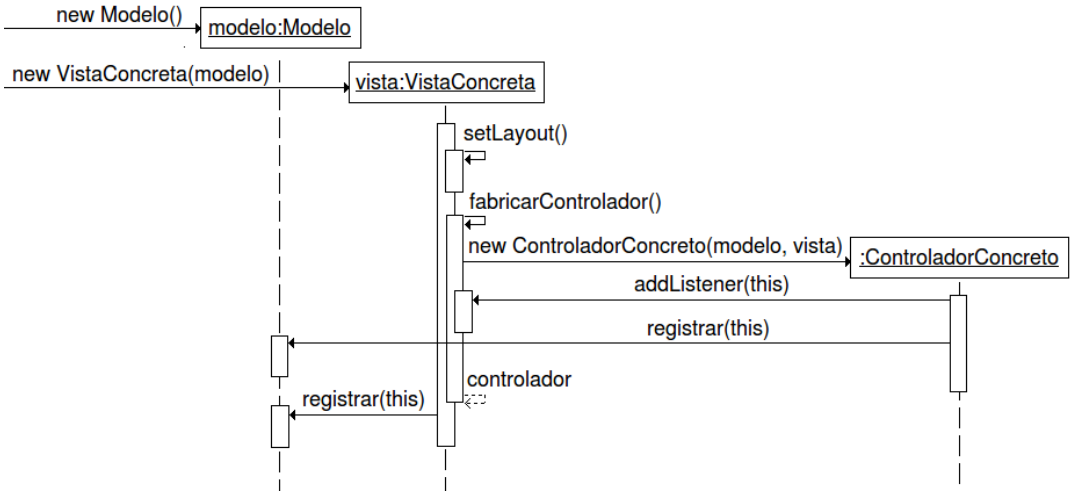

FIGURA 6.2: Cómo el patrón MVC es inicializado.

En la figura 6.2 se muestran los pasos que toma la inicialización de los componentes del patrón MVC. Como primera medida se crea una instancia del modelo, lo que provoca la inicialización de su estado interno. Luego una instancia de una `VistaConcreta` es creada, toma como referencia al modelo creado anteriormente que es pasado como parámetro en el momento de su creación. La vista establece su layout con su método `setLayout()` que fija los tamaños y disposición de los elementos que componen la vista que representará la interfaz de usuario.

En este caso, la vista es la responsable de crear el objeto controlador. Tanto el modelo como la vista misma son pasados como parámetros para la construcción de su controlador. Solo una vista concreta conoce el controlador apropiado para ella, es por eso que debe implementar el método `fabricarControlador()`.

Cuando se crea el `ControladorConcreto`, éste se registra ante la vista para escuchar los eventos que provocan las acciones del usuario que tienen lugar en ella. Con el método `addListener()` el controlador es capaz de escuchar para así poder reaccionar y traducir las acciones del usuario en acciones al modelo o a la vista misma. Si algunas de éstas acciones tiene que tener en cuenta el estado del modelo, el controlador debe registrarse como observador del modelo.

Finalmente, después de que se terminó con la inicialización del controlador, la vista se registra como observador del modelo para mantenerse sincronizada con su estado y poder mostrar información actualizada al usuario.

Otra parte clave en la dinámica del patrón MVC es entender cómo una acción del usuario sobre una vista resulta en cambios del estado del modelo, que dispara el mecanismo de propagación de cambios (Figura 6.3).

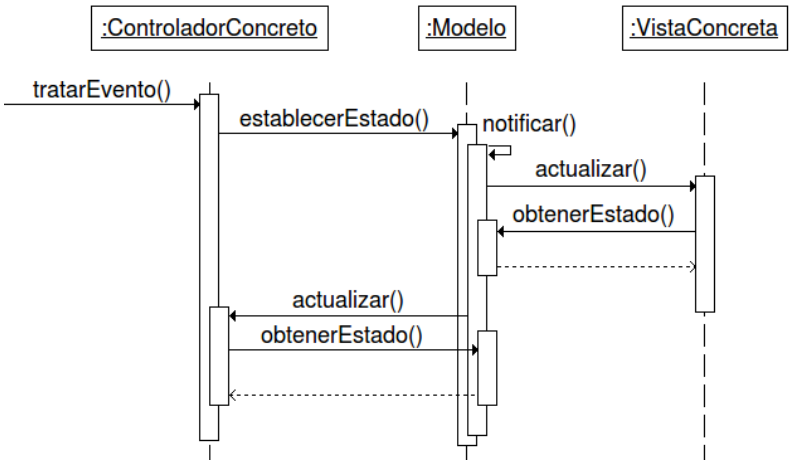

FIGURA 6.3: Sincronización en el patrón MVC.

Cuando el usuario interactúa con la interfaz se generan eventos en el sistema de ventanas de la plataforma. Si el controlador de la vista se halla registrado para escuchar el evento que provocó el usuario, entonces el controlador intercepta este evento, lo interpreta, y activa el servicio correspondiente al evento en el modelo, representado en el diagrama de secuencia como `establecerEstado()`.

El modelo ejecuta el pedido de servicio, y si el brindar este servicio resulta en cambios en su estado, el modelo debe `notificar()` a todas las vistas y controladores registrados en el mecanismo de propagación de cambios. Para sincronizar tanto vistas como controladores, el modelo debe llamar a sus métodos `actualizar()` en cada uno de ellos.

Al recibir la notificación, cada vista pide los datos modificados del estado al modelo y los actualiza en su interfaz gráfica para presentarlos al usuario. Del mismo modo, los controladores obtienen el estado del modelo y verifican si deben modificar alguna funcionalidad de la interfaz gráfica, por ejemplo deshabilitar alguna opción en la vista. La recuperación de los datos actualizados por parte de los observadores se representa de forma esquematizada en la figura 6.3 con el método `obtenerEstado()`.

6.2.6. Ejemplo

Se mostrará una implementación del patrón MVC sobre un ejemplo reducido y esquemático para no perder de vista los mecanismos del patrón al tratar de resolver problemas más reales. Este esquema de aplicación define un framework que puede ser implementado y extendido para cualquier tipo de aplicación interactiva real, como las aplicaciones que cualquier desarrollador podría implementar a lo largo de su carrera.

Supongamos que se necesita una aplicación para llevar la cuenta de alumnos inscriptos en las carreras del departamento de matemáticas. El departamento dicta las carreras de Licenciatura en Matemáticas y la carrera de Profesorado en Matemáticas. Existe un jefe de departamento que se encarga de cuestiones de ambas carreras, y un director para cada carrera que se dicta en el departamento.

El modelo de la aplicación que representará el departamento de matemáticas será tan simple como mantener dos datos numéricos, uno para cada carrera, y brindar operaciones para obtener y establecer[1] sus contenidos. El estado del modelo estará representado solamente por estos dos campos privados, serán del tipo `int` y almacenarán la cantidad de alumnos que cursan en las distintas carreras del departamento.

[1]Corresponden al patrón de implementación o 'Idioms' conocidos como 'Getter and Setter', usados en muchos lenguajes de programación, especialmente en Java. En la clase `Modelo` se usará el método `getAlumnos()` en vez de `obtenerAlumnos()`, y el método `setAlumnos(int a)` en vez de `establecerAalumnos(int a)`.

```
package mvc.modelo;

public class DptoMatematicas{
    private int alumnosLic, alumnosProf;
    public DptoMatematicas(int alumnosLic,int alumnosProf) {
        this.alumnosLic = alumnosLic;
        this.alumnosProf = alumnosProf;
    }
    public void setAlumnosLic(int alumnos) {
        this.alumnosLic = alumnos;
    }
    public void setAlumnosProf(int alumnos) {
        this.alumnosProf = alumnos;
    }
    public int getAlumnosLic() {
        return alumnosLic;
    }
    public int getAlumnosProf() {
        return alumnosProf;
    }
}
```

Código 6.1: Modelo inicial de la aplicación.

Se podría pensar en una mejor implementación para representar los valores numéricos, por ejemplo, con un mapa dinámico, y así hacer referencia a sus contenidos de la manera clave-valor. Esto permitiría agregar y quitar fácilmente nuevas carreras para llevar el conteo de sus alumnos. Pero vamos a concentrarnos en los asuntos y mecanismos referidos al patrón MVC, sin entrar en complicaciones extras.

En el modelo también debe existir toda la lógica funcional de la aplicación referida al dominio del problema que se está tratando de solucionar. En este ejemplo, la lógica es casi inexistente, pero en un problema real el modelo podría estar representado por cientos de clases e interfaces. Entonces, lo que realmente se hizo hasta ahora, al crear la clase `DptoMatematicas`, normalmente a partir de un análisis del dominio del problema de la aplicación, fue separar en el modelo el núcleo funcional de la aplicación de todos los otros mecanismos que serán responsables de la interacción hombre-computador que no responden a requerimientos funcionales. El modelo debe encapsular los datos y funcionalidades necesarias para resolver la lógica fundamental de la aplicación, más que nada relativa al dominio del problema.

Ya separado el modelo, se necesita pensar en los requerimientos no funcionales de la aplicación, como el mecanismo de propagación de cambios que mantendrá sincronizadas las vistas con el estado del modelo. En otras palabras, se necesita aplicar el patrón Observer (Sección 5.3), para luego asignarle el rol de sujeto o publicador a la clase `DptoMatematicas`. Para esto vamos a crear una clase abstracta `Sujeto` que se encargará de mantener todos los observadores y su interfaz ofrecerá métodos para que éstos

puedan suscribirse, eliminarse, y ser notificados de los cambios que ocurran en el estado del modelo. Conjuntamente vamos a definir la interfaz `Observador` a la que deben adherirse los objetos que serán observadores para poder recibir notificaciones del modelo y actualizar su estado. Éste es el primer paso para crear nuestro framework MVC de nuestra aplicación interactiva para el departamento de matemáticas.

```
package mvc;

public interface Observador {
    public abstract void actualizar();
}
```

Código 6.2: Interfaz que deben implementar los observadores.

```
package mvc;

import java.util.ArrayList;

public abstract class Sujeto {
    private ArrayList<Observador> observadores = new ArrayList<Observador>();
    public void registrar(Observador o) {
        observadores.add(o);
    }
    public void quitar(Observador o) {
        observadores.remove(o);
    }
    public void notificarObservadores() {
        for (Observador o : observadores) {
            o.actualizar();
        }
    }
}
```

Código 6.3: Clase abstracta de la que debe heredar el observado.

La clase `DptoMatematicas` debe exponer una interfaz que permita a las vistas obtener el estado y a los controladores manipular los datos del modelo. Se creará una interfaz para separar la implementación del modelo y brindar los métodos necesarios para los observadores.

En una aplicación más compleja es necesario pensar siempre en términos de interfaces para lograr mecanismos más genéricos, aprovechar el polimorfismo, exponer lo justo y necesario de las distintas abstracciones que conforman el modelo del dominio para lograr sistemas flexibles y reutilizables. Con las interfaces básicamente es posible seguir varios principios, como el de programar para una interfaz y lograr un acoplamiento débil. Se dice que el vocabulario del dominio del problema debe estar representado por las distintas interfaces que brinda un sistema.

La interfaz `Modelo` permitirá tanto a vistas como controladores conocer las partes del modelo que les son de interés.

```
package mvc.modelo;
import mvc.Observador;

public interface Modelo {
    public void setAlumnosLic(int alumnos);
    public void setAlumnosProf(int alumnos);
    public int getAlumnosLic();
    public int getAlumnosProf();
    public void registrar(Observador o);
    public void quitar(Observador o);
}
```

CÓDIGO 6.4: Interfaz Modelo.

Siempre es aconsejable organizar los archivos que contienen el código fuente de una aplicación. En este ejemplo se hace uso de la palabra reservada `package` de Java para definir paquetes que se corresponden a una estructura de carpetas en el sistema de archivos. A través de los paquetes se organiza el código fuente y se establecen distintos espacios de nombres. Se separarán los archivos fuentes que implementen las diferentes partes que propone el patrón MVC. Hasta ahora, la estructura de directorios que contiene el código fuente de la aplicación de ejemplo quedaría como se muestra en la Figura 6.4.

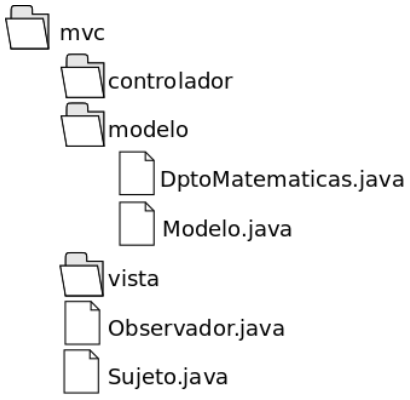

FIGURA 6.4: Estructura de carpetas contenedoras de archivos fuentes.

Ahora es necesario que el objeto que represente al modelo sea capaz de mantener actualizado a sus observadores dentro del mecanismo de propagación de cambios. Para esto la clase `DptoMatematicas` debe heredar de `Sujeto` e incorporar sus capacidades.

Además, también es necesario especificar el momento adecuado en que se deben actualizar los observadores para invocar el método `notificarObservadores()`. El momento adecuado sería después de realizar un cambio en el conteo de la cantidad de alumnos

del departamento en la clase `DptoMatematicas`. Es conveniente controlar los datos que se establecen en el estado para evitar disparar el mecanismo de propagación de cambios innecesariamente, ya que esto puede ser muy costoso en el caso de existir un gran número de objetos observadores. Al momento de establecer el estado del modelo se procederá a validar el dato. Se supone que el estado del modelo mantiene valores numéricos que representan cantidades donde no es admisible un número negativo de alumnos.

```
package mvc.modelo;

import mvc.Sujeto;

public class DptoMatematicas extends Sujeto implements Modelo{
    ...
    public void setAlumnosLic(int alumnos) {
        if (validar(alumnos) && alumnos != this.alumnosLic) {
            this.alumnosLic = alumnos;
            notificarObservadores();
        }
    }
    public void setAlumnosProf(int alumnos) {
        if (validar(alumnos) && alumnos != this.alumnosProf) {
            this.alumnosProf = alumnos;
            notificarObservadores();
        }
    }
    private boolean validar(int dato) {
        return (dato >= 0);
    }
    ...
}
```

Código 6.5: Modelo con el rol de Sujeto.

Los observadores de esta pequeña aplicación MVC serán las vistas y controladores, que deberán ser notificados ante cambios en el estado del modelo. Como esto se trata de un framework, se definirán clases abstractas que representen el comportamiento fundamental para todas las vistas y los controladores, a partir de las cuales se deberá heredar para definir vistas y controladores concretos.

La clase abstracta `Vista` implementará la interfaz `Observador` para definir su rol de observador en el framework y poder reflejar los cambios del estado del modelo en las vistas concretas. Será responsabilidad de las vistas concretas implementar el método `actualizar()` de la interfaz `Observador` para incorporar el procedimiento correspondiente al momento en que el modelo envíe una notificación a través del mecanismo de propagación de cambios.

```java
package mvc.vista;

import mvc.Observador;
import mvc.modelo.*;
import mvc.controlador.*;

public abstract class Vista implements Observador {
    private Modelo modelo;
    private Controlador controlador;

    public Vista(Modelo modelo) {
        this.modelo = modelo;
        modelo.registrar(this);
        this.setLayout();
        this.controlador = fabricarControlador();
        this.actualizar();
    }
    protected abstract void setLayout();
    protected Controlador fabricarControlador() {
        return null;
    }
    public Controlador getControlador() {
        return controlador;
    }
    public Modelo getModelo() {
        return modelo;
    }
}
```

Código 6.6: Vista con el rol de Observador.

El constructor de la clase abstracta `Vista` comienza el proceso de inicializar una vista, para ello almacena una referencia al modelo, registra la vista como observador, llama al método `setLayout()` que debe ser implementado en cada vista concreta, guarda una referencia al controlador adecuado según la vista concreta, y finalmente llama al método `actualizar()` para mostrar la información actualizada del estado del modelo en la vista.

La clase abstracta `Vista` define el comportamiento base que debe tener una vista concreta, pero no sabe nada sobre cómo se disponen sus elementos visuales. Es por esto que las vistas concretas deben implementar obligadamente el método `setLayout()` para definir todos los elementos visuales que la vista concreta contendrá.

La vista abstracta tampoco sabe sobre cuál es el controlador adecuado para una vista concreta, pero sí sabe que en el momento de inicialización debe almacenar una referencia a éste controlador. Se deja que la vista concreta, de ser necesario, sobreescriba `fabricarControlador()` para que retorne la referencia adecuada a su controlador específico. Con el método `fabricarControlador()` se está aplicando el patrón Method Factory (Sección 5.2).

Para crear un controlador es necesario pasar como parámetros a su constructor tanto el modelo como la vista. Al igual que la clase `Vista`, la clase abstracta `Controlador` se registra como observador del modelo y luego llama al método `actualizar()`. Esta vez se implementa un procedimiento vacío por defecto para el método `actualizar()`, ya que para esta aplicación los controladores concretos no reaccionan ante un cambio en el modelo, sólo reaccionan ante la interacción del usuario con las vistas[2].

```
package mvc.controlador;

import mvc.Observador;
import mvc.modelo.*;
import mvc.vista.*;

public abstract class Controlador implements Observador {
    private Modelo modelo;
    private Vista vista;
    public Controlador(Modelo modelo, Vista vista) {
        this.vista = vista;
        this.modelo = modelo;
        modelo.registrar(this);
        actualizar();
    }
    public void actualizar() {}
    protected Modelo getModelo() {
        return modelo;
    }
    protected Vista getVista() {
        return vista;
    }
}
```

Código 6.7: Controlador con el rol de Sujeto.

Habiendo creado las clases abstractas `Sujeto`, `Vista`, `Controlador`, y las interfaces `Observador` y `Modelo`, queda implementado un framework MVC para la aplicación de ejemplo, que facilita la creación de distintas vistas, cada uno con su respectivo controlador, donde sin grandes esfuerzos de programación podrán mantener sus estados sincronizados con el estado del modelo que representa el departamento de matemáticas.

Ahora, supongamos que el jefe del departamento de matemáticas necesita de una interfaz gráfica que le permita establecer la cantidad de alumnos que se va contando, tanto en el profesorado como en la licenciatura de matemáticas. Esto resulta en definir una vista concreta para el modelo, que muestre el contenido de los datos numéricos que el modelo almacena, y presente al usuario cada dato en un área de texto donde pueda escribir

[2]Sería válido pensar en controladores que directamente cuenten con la opción de registrarse, o no, como observador del modelo, pero se está construyendo un framework simplificado y a la vez algo genérico para la aplicación de ejemplo.

un nuevo valor a establecer. Para que el nuevo valor quede almacenado en el modelo, el usuario debe, por ejemplo, presionar la tecla *enter*. Esta vista concreta debe estar preparada para que refleje el estado actual de la cantidad de alumnos en cada carrera, en el caso de que el conteo cambie desde otra vista que el departamento podría requerir. De esta forma se puede brindar al jefe del departamento la interacción necesaria para visualizar y establecer la cantidad de alumnos por carrera en su departamento.

```
package mvc.vista;
...
import mvc.modelo.*;
import mvc.controlador.*;

public class VistaTabla extends Vista {
    private JFrame ventana;
    private JTextField alumnosLic, alumnosProf;
    private JLabel etiquetaLic, etiquetaProf;
    private JPanel panel;

    public VistaTabla(Modelo modelo) {
        super(modelo);
    }
    @Override
    protected Controlador fabricarControlador() {
        return new ControladorTabla(getModelo(), this);
    }
    @Override
    public void actualizar() {
        alumnosLic.setText(Integer.toString(getModelo().getAlumnosLic()));
        alumnosProf.setText(Integer.toString(getModelo().getAlumnosProf()));
    }
    protected void setLayout() {
        ventana = new JFrame("Departamento Matematicas");
        //codigo que crea y establece los elementos de la vista
        ...
        ventana.setVisible(true);
    }
    public String getAlumnosLic() {
        return alumnosLic.getText();
    }
    public String getAlumnosProf() {
        return alumnosProf.getText();
    }
    public void addAlumnosLicListener(ActionListener controller) {
        alumnosLic.addActionListener(controller);
    }
    public void addAlumnosProfListener(ActionListener controller) {
        alumnosProf.addActionListener(controller);
    }
}
```

Código 6.8: Una vista concreta.

`VistaTabla` va a necesitar de un controlador que interprete la acción de presionar *enter* en una acción al modelo. Necesitamos implementar `ControladorTabla` como especifica el método `fabricarControlador()` de nuestra vista concreta. Se ha de notar que `VistaTabla` presenta en su interfaz el método `addAlumnosLicListener(...)`, que permitirá al controlador escuchar los eventos levantados por el usuario cuando presione la tecla *enter* en el área de texto. De esta forma se brinda una operación para que el controlador se registre para escuchar los eventos de un elemento de la vista.

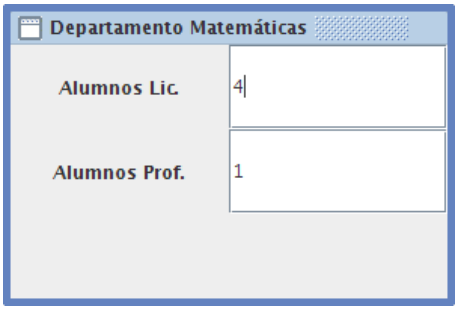

FIGURA 6.5: Tabla de cantidad de alumnos por carrera.

Por cada vista se especifica el comportamiento de la aplicación a través de su controlador en respuesta a las acciones del usuario. Cuando el usuario presione *enter* en el área de texto para cada cantidad de alumnos de la `VistaTabla`, debe establecerse el nuevo valor en el modelo por medio de su controlador, lo que provocará la activación del mecanismo de propagación de cambios y actualizará todos los observadores existentes.

```
package mvc.controlador;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import mvc.modelo.*;
import mvc.vista.*;

public class ControladorTabla extends Controlador{
    public ControladorTabla(Modelo modelo, Vista vista) {
        super(modelo, vista);
        VistaTabla vistaTabla = (VistaTabla) vista;
        vistaTabla.addAlumnosLicListener(new AlumnosLicControlador());
        vistaTabla.addAlumnosProfListener(new AlumnosProfControlador());
    }
    private class AlumnosLicControlador implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            VistaTabla vt = (VistaTabla) getVista();
            getModelo().setAlumnosLic(Integer.parseInt(vt.getAlumnosLic()));
        }
    }
    private class AlumnosProfControlador implements ActionListener {
        public void actionPerformed(ActionEvent e) {
```

```
            VistaTabla vt = (VistaTabla) getVista();
            getModelo().setAlumnosProf(Integer.parseInt(vt.getAlumnosProf()));
        }
    }
}
```

Código 6.9: Controlador concreto.

`ControladorTabla` hereda de `Controlador`, y recibe un modelo y una vista para su construcción. El controlador concreto conoce a su vista y registra ante ella sus clases internas para escuchar los eventos del usuario. `ControladorTabla` define estas clases internas que se encargarán de tratar los eventos que sucedan en el área de texto.

Para tener una visión más general de cómo se reparten las cantidades de alumnos entre las carreras, el jefe de departamento desea incluir en la aplicación una vista que muestre la cantidad de alumnos del departamento en un gráfico de torta, discriminando la carrera de Licenciatura del Profesorado. Para realizar esto debemos crear una nueva vista concreta.

```
package mvc.ejemplo.vista;
...
import mvc.modelo.*;

public class VistaTorta extends Vista {
    private JFrame ventana;
    private JFreeChart jfc;
    private DefaultPieDataset dataset;

    public VistaTorta(Modelo modelo) {
        super(modelo);
    }
    protected void setLayout() {
        ...
        ventana.setVisible(true);
    }
    @Override
    public void actualizar() {
        dataset.setValue("Lic.", getModelo().getAlumnosLic());
        dataset.setValue("Prof.", getModelo().getAlumnosProf());
    }
}
```

Código 6.10: Otra vista para el modelo.

Como esta vista no permite modificar la cantidad de alumnos, se puede decir que es una vista de solo lectura, no es necesario implementar un controlador para la misma, por lo tanto la implementación predeterminada de `fabricarControlador()` que brinda la clase abstracta `Vista` es suficiente. Para poder graficar el gráfico de torta se usa la API JFreeChart[3], la cual facilita el dibujo de distintos tipos de diagramas.

[3] JFreeChart es una librería open source http://www.jfree.org/jfreechart.

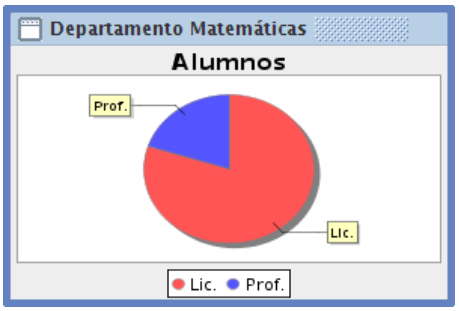

FIGURA 6.6: Diagrama de torta, como se reparte los alumnos en las carreras.

Supongamos ahora, que el director de la carrera de Licenciatura en Matemáticas requiere un GUI donde pueda hacer correcciones sobre el conteo de alumnos inscriptos en la carrera que dirige. Esto es una interfaz específica para la dirección de la carrera de Licenciatura, que presenta la cantidad de alumnos y es posible interactuar y modificar esa cantidad en cualquier momento.

```
package mvc.vista;
...
import mvc.controlador.*;
import mvc.modelo.*;

public class VistaLicenciatura extends Vista {
    private JFrame ventana;
    private JLabel etiquetaLic;
    private JSpinner spinnerLic;
    private JPanel panel;

    public VistaLicenciatura(Modelo modelo) {
        super(modelo);
    }
    protected void setLayout() {
        ...
        ventana.setVisible(true);
    }

    @Override
    protected Controlador fabricarControlador() {
        return new ControladorLicenciatura(getModelo(), this);
    }
    public int getAlumnosLicenciatura() {
        return (Integer) spinnerLic.getModel().getValue();
    }

    @Override
    public void actualizar() {
        spinnerLic.setValue(getModelo().getAlumnosLic());
    }
```

```
    public void addListener(ChangeListener controller) {
        spinnerLic.addChangeListener(controller);
    }
}
```

Código 6.11: Otra vista para el modelo.

Como esta vista permite la interacción del usuario para modificar la cantidad de alumnos en la carrera, se debe implementar un controlador concreto para que interprete los eventos de la vista. Hay que crear la clase `ControladorLicenciatura`, que simplemente sabrá como responder a las acciones del usuario sobre la vista.

```
package mvc.controlador;
...
import mvc.modelo.*;
import mvc.vista.*;
public class ControladorLicenciatura extends Controlador {
    public ControladorLicenciatura(Modelo modelo, Vista vista) {
        super(modelo, vista);
        VistaLicenciatura vl = (VistaLicenciatura) vista;
        vl.addListener(new ControladorAlumnos());
    }
    private class ControladorAlumnos implements ChangeListener {
        public void stateChanged(ChangeEvent e) {
            VistaLicenciatura vl = (VistaLicenciatura) getVista();
            getModelo().setAlumnosLic(vl.getAlumnosLicenciatura());
        }
    }
}
```

Código 6.12: Otra vista para el modelo.

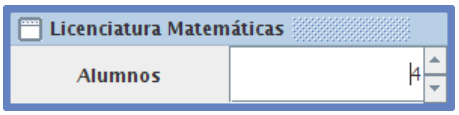

Figura 6.7: Vista para la dirección de la carrera de licenciatura.

Hemos logrado crear diferentes vistas de un mismo modelo según la necesidad de los distintos usuario. Esta tarea fue sencilla dado que una vez que disponemos del framework MVC solo nos debemos preocupar por cuestiones propias de las vistas y de la interacción del usuario. Si fuese necesario también podríamos crear una vista que muestre las cantidades de alumnos por carrera en un diagrama de barras, otra vista para uso exclusivo en la dirección de la carrera del profesorado, etc.

Para poner en funcionamiento la aplicación, primero se creará una ventana con opciones para manejar las diferentes vistas del modelo, esto nos permitirá crear y desechar interfaces gráficas de usuario en tiempo de ejecución. La ventana estará representada por una

clase que se encargará de crear el modelo, y dispondrá de un método de creación que decidirá qué vista crear según la vista que se haya seleccionado. Esta ventana presentará al usuario una serie de elementos gráficos *checkbox*[4] para seleccionar la vista del modelo con queramos ver. Si bien esta ventana representará la aplicación, la misma no refleja ninguna parte del modelo, y por lo tanto no utilizará los mecanismos del framework.

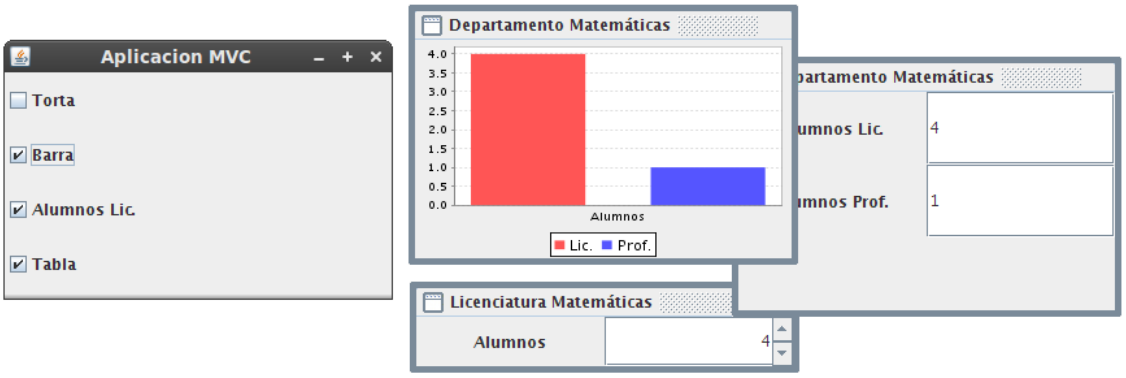

FIGURA 6.8: Interfaces de la aplicación MVC.

La clase `Aplicacion` representa la ventana de la aplicación con la cual es posible presentar y ocultar diferentes vistas del modelo del departamento de matemáticas:

```
package mvc;
...
import mvc.modelo.*;
import mvc.vista.*;
public class Aplicacion {
    private Modelo modelo;
    private JFrame ventana;
    private JPanel panel;
    private JCheckBox torta, barra, alumnosLic, tabla;
    public Aplicacion() {
        modelo = new Modelo(4, 1); //se crea el modelo
        setLayout();
    }
    protected void setLayout() {
        //crea los checkbox para cada vista y el Listener
        //para escuchar los eventos de los checkboxes.
        ventana.setVisible(true);
    }
    class Listener implements ItemListener {
        private Vista vistaBarra, vistaTorta, vistaSimple, vistaTabla;
        public void itemStateChanged(ItemEvent e) {
            //selecciona la vista a crear o la vista a desechar por evento
        }
        private Vista crearVista(ItemSelectable itemSelectable) {
```

[4]'Checkbox' es un elemento de una interfaz de gráfica que permite al usuario hacer múltiples selecciones.

```
            Vista v = null;
            if (itemSelectable == barra) {
                v = new VistaTorta(modelo);
            } else if (itemSelectable == torta) {
                v = new VistaBarra(modelo);
            } else if (itemSelectable == tabla) {
                v = new VistaTabla(modelo);
            } else if (itemSelectable == alumnosLic) {
                v = new VistaLicenciatura(modelo);
            }
            return v;
        }
    }
}
```

Código 6.13: Ventana para seleccionar vistas del modelo.

Lo único que hace falta ahora es instanciar la clase `Aplicacion` para poder ver en funcionamiento la aplicación[5].

Hemos creado vistas desacopladas del modelo al establecer entre ellos un protocolo de registración notificación para asegurarnos de que las GUIs reflejen el estado actual del modelo. De una interfaz gráfica de usuario se separaron las cuestiones intrínsecas de la parte visual de los aspectos de su interacción en la dupla vista-controlador, también se ordenó el código fuente en una estructura de directorios, y se logró independizar el modelo. Todo esto es posible gracias a la organización de los distintos elementos al aplicar una arquitectura MVC en nuestra aplicación.

6.3. Principios y patrones

El principio fundamental que se aplica en el patrón arquitectónico MVC es el principio de separación de intereses (Sección 3.2.7). Aplicando este principio se logra separar el dominio del problema, la visualización de la aplicación, y las entradas del usuario en partes especializadas.

En el ejemplo que se mostró se puede apreciar los patrones que componen el framework MVC del sistema, los principios que aportan, y los problemas que solucionan.

6.3.1. Patrón Observer

El patrón Observer (Sección 5.3) soluciona el problema de mantener sincronizado el estado del modelo con la vista, y el controlador, si fuese necesario. También, mantiene

[5]También es necesario hacer pequeños ajustes en las vistas del modelo para que sea posible abrirlas y cerrarlas desde la ventana de la aplicación.

débilmente acoplado el dominio de la aplicación de las cuestiones referidas a la presentación e interacción con el usuario. Gracias al patrón Observer podemos crear diferentes vistas de un mismo modelo, hasta mostrar todas las vista al mismo tiempo y que permanezcan actualizadas.

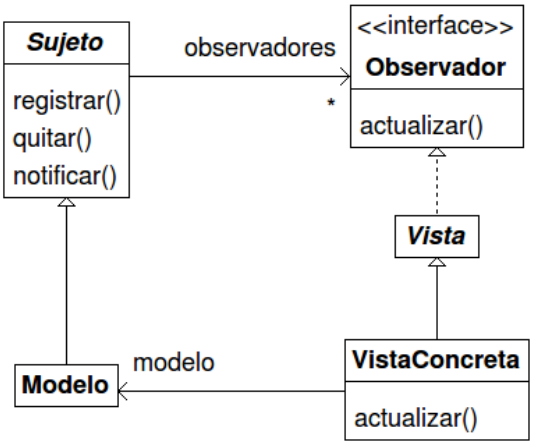

FIGURA 6.9: Patrón Observador en ejemplo MVC.

En el ejemplo del patrón MVC que se mostró no se necesita actualizar el controlador, por eso se dio una implementación por defecto del método `actualizar()` en la clase abstracta `Controlador`. Pero podría ser necesario sobreescribir el método si, por ejemplo, el modelo sólo permitiese ingresar la cantidad de alumnos hasta que la suma de la cantidad ingresada alcance un total conocido. Si fuese así, el modelo debería mantener un contador de los alumnos que se va ingresando, y cuando alcance el total conocido de alumnos notificar lo sucedido para que el controlador responda deshabilitando la posibilidad del usuario de interactuar con la vista para no permitir seguir ingresar más alumnos, o tal vez mostrar un mensaje.

6.3.2. Patrón Strategy

Las vistas y controladores implementan el patrón Strategy (Sección 5.1). El controlador representa el comportamiento de la vista con respecto al modelo de la aplicación, y podría ser fácilmente intercambiado por otro controlador si se necesitase un comportamiento diferente para una sola vista, por ejemplo asignar un controlador nulo a la vista para que no sea posible ninguna interacción. La vista es un objeto que se configura con una estrategia de comportamiento brindada por el controlador, sólo es responsable de presentar la información, y delega en el controlador todas las decisiones acerca de qué operaciones realizar en el modelo. El patrón Strategy también ayuda a mantener el

modelo desacoplado de la vista, porque es el controlador el responsable de traducir las acciones del usuario en llamadas al modelo.

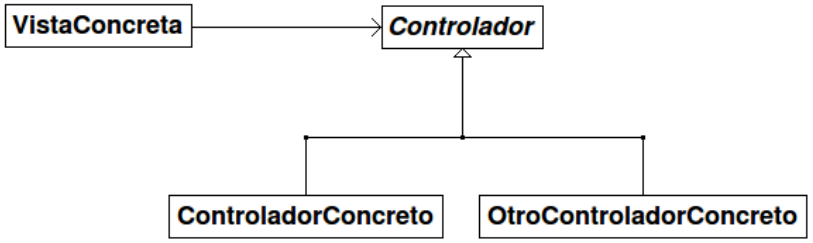

FIGURA 6.10: Patrón Strategy en ejemplo MVC.

6.3.3. Patrón Method Factory

Cada vista concreta especifica qué controlador concreto crear para su propia configuración, esta decisión es realizada en las subclases de la clase abstracta `Vista`. El framework provee los mecanismos y sabe el momento adecuado en que una vista debe configurarse con un controlador, pero no sabe nada acerca qué controlador crear, el desarrollador debe especificar el controlador concreto para cada vista sobreescribiendo el método `fabricarControlador()`. Esto es un ejemplo de aplicación del patrón Factory Method cuya estructura se explicó en la Sección 5.2.

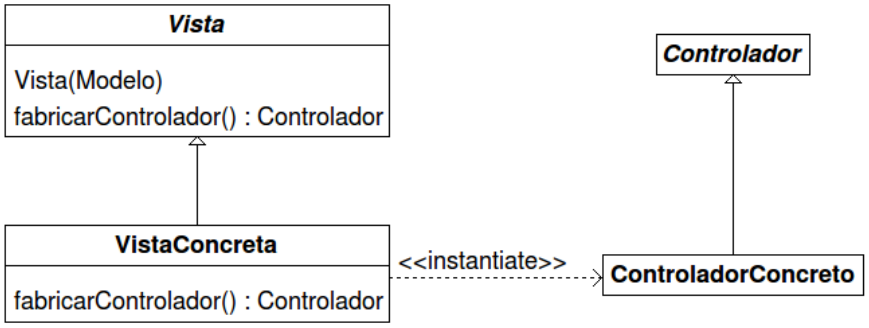

FIGURA 6.11: Patrón Method Factory en ejemplo MVC.

6.3.4. Patrón Composite

En las vistas de las GUIs, dependiendo de la plataforma gráfica que estemos usando, se puede decir que internamente se hace uso del patrón Composite para administrar los diferentes componentes que contiene una ventana.

Casi todos los sistemas de ventanas, como por ejemplo Swing de Java, son bastantes sofisticados, y es difícil de notar en su estructura interna el uso que hacen del patrón Composite (Sección 5.4). Una ventana es un contenedor de los elementos de una GUI, como botones, área de texto, y etiquetas, que actúa como el componente más alto en la jerarquía de la estructura de árbol de todos los hijos que contiene. Estos elementos a su vez pueden ser elementos compuestos como un panel, que contienen otros elementos. Con el patrón Composite la tarea del desarrollador se ve simplificada, sólo debe hacer uso de los componentes gráficos que la librería brinda para armar su interfaz y despreocuparse del modo en que se dibujan.

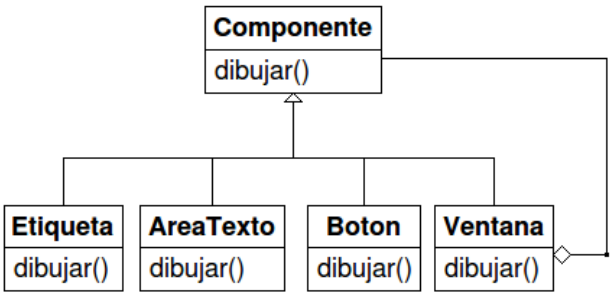

FIGURA 6.12: Patrón Composite en ejemplo MVC.

En los tiempos en que MVC recién se empezaba a usar, crear una GUI requería mucho más trabajo en cuanto a la programación de los mecanismo que hacía posible representación gráfica. La aplicación del patrón Composite era más directa y visible al momento de trabajar con interfaces de usuario, y no estaba escondida en una librería como sucede en la actualidad.

Capítulo 7

Conclusión

Se trataron temas que van desde principios elementales relacionados de manera más estrecha con la programación, hasta temas relacionados con el diseño, arquitectura, desarrollo, e ingeniería del software. Nada de lo que se vio es una simple guía de pasos a seguir, requiere cargar nuestro cerebro de herramientas conceptuales, no solamente mediante el estudio, sino primordialmente mediante la práctica y experiencia.

Varios temas fueron desarrollados, pero no se mostraron muchas variaciones sobre los mismos temas. Esencialmente cuando más variaciones de un patrón conozcamos, más preciso será el concepto que alcanzaremos sobre ese patrón. La realidad es que todo de lo que se esta hablando y exponiendo en este trabajo, tiene su origen, y trata el problema sobre cómo lidiar con el cambio que siempre acecha a un sistema de información.

7.1. Aplicación del patrón MVC

El patrón Modelo Vista Controlador, más allá de ser un patrón que encapsula conocimiento y experiencia, que nos sirve de herramienta para aplicar principios de manera eficiente, también, por ser un patrón arquitectónico, está fuertemente relacionado con las decisiones de diseño de la arquitectura en un sistema, y es ahí donde podemos encontrar cómo el aplicar este patrón se relaciona, influye, y permite al mismo tiempo una flexibilidad en la ingeniería de software que utilizamos durante el desarrollo.

Aplicar con conciencia el patrón MVC es resolver la ecuación de la Figura 7.1. Tener siempre presente los *principios* nos sirve de guía para resolver los problemas mediante la experiencia acumulada en los *patrones*. Al resolver problemas arquitectónicos, como el problema de la implementación de las GUIs para aplicaciones interactivas, con MVC estamos tomando decisiones de diseño con un alto nivel de abstracción sobre la *arquitectura* del sistema que van a permitir la aplicación de una *ingeniería* de software, para traer principalmente orden, facilitar el trabajo en paralelo del equipo, y posibilitar una aplicación más cómoda de una metodología de desarrollo.

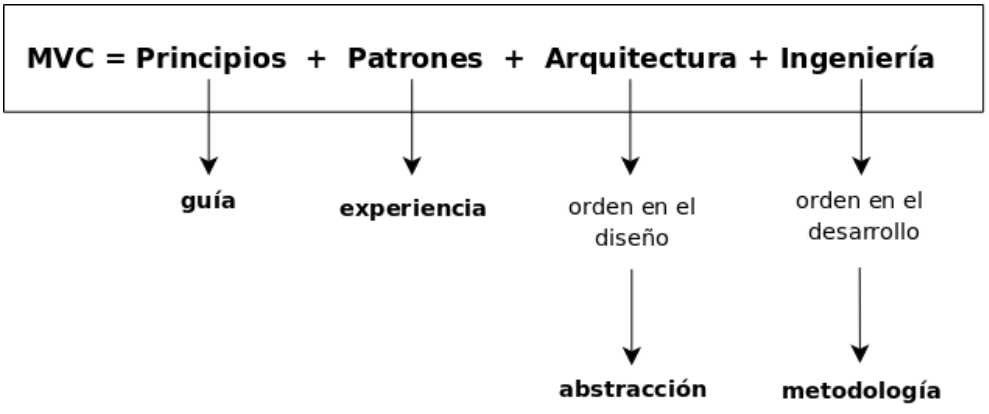

FIGURA 7.1: MVC es más que principios y patrones.

- Principios
  Los principios aplicados en forma aislada y esporádicamente no tienen mucho sentido. Todos los principios deben estar presentes a la hora del diseño e implementación de un sistema, y más aún en el momento de adaptar un patrón al problema en particular que deseamos resolver.

- Patrones
  Los patrones aplicados como si fuesen recetas pierden toda su fortaleza, y pueden

llevarnos a peores resultados que no aplicarlos. En un patrón compuesto es necesario saber cómo los patrones que lo componen colaboran entre sí para lograr el propósito o fin que se proponen. Esto es conocer el rol de cada una de sus partes, y cómo fluye la comunicación entre estas partes para lograr su correcta aplicación.

- Arquitectura
  Desconocer o no definir en términos generales la arquitectura de un sistema lleva su desarrollo al caos, a la imposibilidad de seguir creciendo y evolucionar, y dificulta la incorporación de una metodología de desarrollo en forma adecuada.

- Ingeniería
  No seguir una metodología en el desarrollo de un sistema ocasiona la perdida del control de su desarrollo. No seremos capaces de hacer estimaciones precisas, y no podremos conocer el riesgo de afrontar un sistema.

Desconocer los principios que involucra, los patrones que colaboran, cómo influye su aplicación en el orden del sistema al determinar una arquitectura, y cómo facilita la ingeniería de software, convierte al patrón MVC en una ecuación imposible de resolver.

7.2. Abstracción, granularidad, y complejidad

Los patrones que se mostraron fueron clasificados tanto por su propósito o intención que persiguen, como por su granularidad. Es la clasificación bidimensional que usa Bushmman et ál. [BMR+96].

En el Cuadro 7.1 la primera columna corresponde al propósito o categoría de problema, y la primera fila al grado de granularidad que posee el patrón. Ésta es una clasificación bidimensional ya que usa dos criterios.

Cuadro 7.1: Clasificación de los patrones vistos.

-	Patrones De Arquitectura	Patrones De Diseño	Patrones De implementación
Sistemas Interactivos	MVC		
Comunicación		Observer	
Variación de Servicio		Strategy	
Descomposición Estructural		Composite	
Creación			Factory Method

Un patrón arquitectónico es clasificado con un mayor nivel de granularidad que un patrón de implementación. Un patrón arquitectónico posee un nivel de granularidad amplio ya que sólo es posible divisarlo con claridad desde un alto nivel de abstracción. Un patrón de implementación, se dice, tiene un nivel de granularidad más fino. A mayor abstracción es posible detectar las partes más gruesas del sistema.

Otro enfoque en cuanto a la granularidad, sin tener en cuenta el nivel de abstracción, es cuán fina son las partes de un sistema en cuanto a la implementación. Por ejemplo, un sistema tendrá un bajo grado de granularidad si solo está compuesto de unas cuantas clases grandes, y tendrá un alto grado de granularidad si posee muchas clases pequeñas[1].

- Granularidad
  Lo que determina la complejidad de un sistema termina siendo el nivel de granularidad que existe en el mismo. Cuando más separado en pequeñas partes, más granulado será el sistema, y por lo tanto más complejo, pero a la vez más flexible y preparado para el cambio. Los patrones agregan complejidad a los sistemas, pero a la vez brindan características que no pueden faltar en ningún sistema serio y de envergadura.

- Abstracción
  La abstracción siempre es un problema, especialmente en la comunicación. Nunca es fácil determinar los límites de los diferentes niveles de abstracción que manejamos, existen solapamientos, y sus fronteras son muchas veces difusas. Pero la abstracción es la herramienta primordial para atacar la complejidad. Diferentes niveles de abstracción bien especificados nos permiten manejar con más soltura los diferentes niveles de granularidad existentes en un sistema.

Manejar correctamente el nivel de abstracción y granularidad que debe poseer un sistema, sin una complejidad innecesaria y con una flexibilidad adecuada, es saber balancear estas necesidades para producir un producto que se adapte al contexto en donde el emprendimiento de software se genera y evoluciona. Este balance es a la vez es una de las claves del éxito del software.

[1] Se dice que un sistema es "coarsed-grained" si consiste de unos pocos pero grandes componentes, y "fine-grained" si esta compuesto de muchos componentes pequeños.

7.3. Esfuerzo de desarrollo

Si encaramos la creación de un sistema aplicando una arquitectura MVC, debemos realizar un esfuerzo inicial mayor que el de comenzar un sistema sin pensar en todos los mecanismos para lograr la flexibilidad que ofrece el patrón.

Podremos obtener resultados inmediatos al comienzo de la creación de un sistema si no planteamos su arquitectura ni pensamos en su flexibilidad, pero a la larga, el esfuerzo de extender y mantener el software será mayor, y por lo tanto más costoso. De esta forma sería casi imposible entregar software de calidad en tiempo y forma.

La decisión de implementar o no una arquitectura MVC depende de la complejidad del sistema que vamos a desarrollar. La Figura 7.2 ayuda a interpretar la relación que existe entre el esfuerzo de desarrollo y la complejidad del sistema.

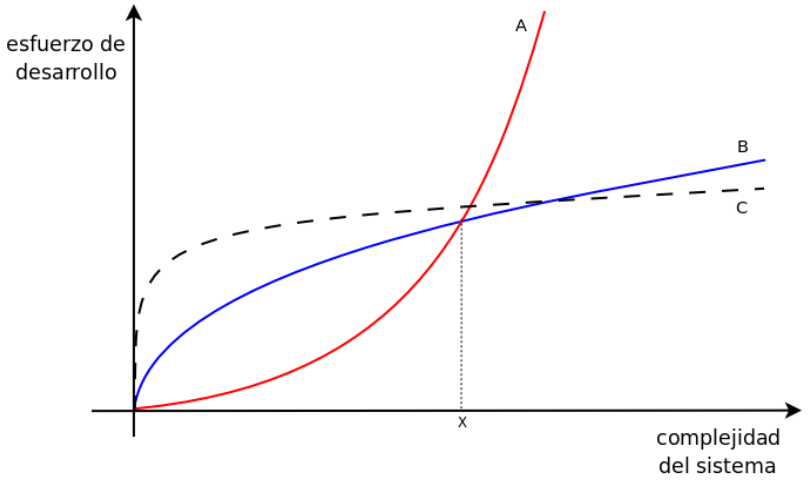

FIGURA 7.2: Complejidad vs Esfuerzo.

- La curva *"A"* representa la relación esfuerzo-complejidad que puede existir en un sistema sin la implementación de una arquitectura MVC. Se puede ver que un sistema de escasa complejidad puede ser desarrollado con un mínimo esfuerzo, pero el desarrollo de un sistema de gran complejidad resulta en un esfuerzo imposible de afrontar de esta manera.
- La curva *"B"* dibuja esta relación con las ventajas de implementar el patrón arquitectónico MVC. Es posible observar que de esta manera es factible afrontar el desarrollo de un sistema de una complejidad considerable por todos los beneficios que brinda el patrón.

En este gráfico, también es válido considerar al eje de *complejidad del sistema* como una línea de tiempo, ya que a medida que el tiempo pasa, podemos decir que el sistema crece en complejidad mientras lo desarrollamos. También sería válido considerar el *esfuerzo de desarrollo* como la cantidad de tiempo que nos tomaría desarrollar una aplicación de una cierta complejidad.

Según el gráfico, se puede decir que si estimamos que la complejidad de un proyecto puede llegar a superar *"X"*, entonces deberíamos considerar seriamente la implementación del patrón arquitectónico MVC si es que no queremos realizar un esfuerzo innecesario en el desarrollo.

Cuanto más nos alejemos de *"X"*, donde la "$complejidad > X$", podemos situar los sistemas que están pensados para tener una interacción intensiva con varios tipos de usuarios que requieren de diferentes vistas del sistema en forma simultanea.

Del otro lado tenemos una "$complejidad < X$" para sistemas que generalmente no necesitan de una interfaz interactiva de usuario, por lo tanto no merece el esfuerzo aplicar una arquitectura MVC.

En las inmediaciones de *"X"*, en el punto donde las curvas se cruzan, puede pensarse en la complejidad de una aplicación interactiva para un solo usuario con unas pocas interfaces para interactuar con el sistema. Un sistema interactivo de escasa complejidad.

Siempre hay que tener en cuenta la evolución que el sistema puede sufrir, aun más cuando hablamos de aplicaciones interactivas porque estamos sujetos a nuevas funcionalidades, y cambios que los usuarios pueden requerir por su uso directo y constante de la aplicación a través de sus interfaces. Hay que recordar que el código de las interfaces de usuarios es siempre propenso al cambio.

7.3.1. Frameworks

Un framework es un sistema de software parcialmente terminado. Define una arquitectura y provee bloques constructivos básicos para la construcción de sistemas. También define donde deben ser hechas las adaptaciones para funcionalidades específicas que brinda. En un ambiente orientado a objetos, un framework consiste en clases abstractas y concretas. [BMR+96]

Si no queremos vernos envuelto en la complejidad de crear desde cero una arquitectura MVC, podemos optar por usar unos de los tantos frameworks MVC que determinan la arquitectura y a la vez nos brindan librerías de utilidades. Por lo general, los frameworks

MVC disponibles actualmente también solucionan otros aspectos de un sistema, como la seguridad, las sesiones, la internacionalización, etc.

Aprender a usar un framework requiere de un tiempo hasta que entendamos los mecanismos que nos brinda. Si nunca hemos usado algún framework necesitaríamos invertir mucho esfuerzo en el comienzo, como se muestra en la curva *"C"* de la Figura 7.2, pero al tiempo que nuestra aplicación crezca en complejidad podremos ver los beneficios por todos los aspectos que el framework soluciona.

La relación esfuerzo-complejidad también se verá determinada por, entre otras cosas, la capacidad y experiencia del equipo de desarrollo que usa el framework. Es necesario destacar que distintos frameworks resultarían en diferentes curvas de relaciones esfuerzo-complejidad para el desarrollo de un sistema. No existe una curva para todos los tipos de frameworks, ni para todos los tipos de desarrolladores.

7.4. Vocabulario común

En la introducción se dijo que desafortunadamente muchas veces tomamos un vocabulario que no nos pertenece y creamos indefectiblemente un problema en la comunicación. No se trata de comunicarnos en términos de patrones porque sí, debemos ser capaces de abarcar y capturar su concepto al momento de comunicarnos a través de ellos.

El valor de un vocabulario común en un grupo de desarrollo es muy importante, ya que cuando se desarrolla software en equipo, la comunicación es uno de los factores que más influye en el éxito de un proyecto.

Cuando nos comunicamos con patrones, no solo comunicamos el nombre del patrón, sino comunicamos conjuntos de características, restricciones, y requerimientos en una sola palabra. Comunicarse con patrones permite que otros desarrolladores capten de inmediato y de manera precisa lo que estamos diciendo. Comunicándonos con patrones podemos decir que menos es más.

7.5. Separación de roles

Si bien el patrón MVC plantea una separación en tres partes, es posible ver dos separaciones principales: la separación de la presentación del modelo, y la separación de la vista del controlador. En la Figura 7.3 se remarca con una línea punteada el rol de presentación, que está compuesto en conjunto por el controlador y la vista.

El modelo y la presentación tratan aspectos totalmente diferentes en un sistema. Cuando desarrollamos el modelo, pensamos en la lógica de negocio y tal vez la persistencia en base de datos. Cuando trabajamos en la presentación pensamos en los mecanismos de una interfaz de usuario.

La clave de esta separación esta dada en la dirección de las dependencias entre las partes, representadas con flechas también en la Figura 7.3. Tanto la vista como el controlador, o sea la presentación, dependen del modelo, pero el modelo no depende de la presentación. Los desarrolladores encargados de programar el modelo pueden estar totalmente despreocupados sobre qué presentación se va a usar. Esta independencia del modelo es posible gracias al acoplamiento débil que se logró usando el patrón Observer.

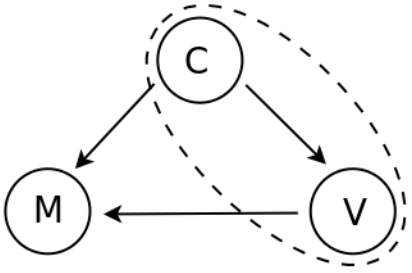

FIGURA 7.3: Modelo, presentación y dependencias esenciales en el patrón MVC.

La separación de la vista del controlador es menos obvia. La razón de esta separación subyace en la posibilidad de poder intercambiar controladores de una misma vista, por ejemplo una vista con comportamiento editable y no editable, donde los controladores son estrategias. En la práctica, la mayoría de las vistas tienen un único controlador y esta separación no es realizada, excepto en interfaces web donde se ve la utilidad de esta separación por las características propias del ambiente web.

La dependencias del modelo, vista, y controlador, quedan determinadas por como fluye la comunicación entre estas partes. El modelo no tiene porqué saber nada de la vista, ni del controlador, sólo necesita informarles que un cambio ocurrió en su estado. Esta comunicación indirecta y mínima, a través de un mecanismo de propagación queda reflejada en la Figura 7.4 por medio de las líneas punteadas.

La vista, y algunas veces el controlador, se registran ante el modelo para poder actualizar sus estados gracias al mecanismo de propagación de cambios. Además, el controlador se registra ante la vista para escuchar los eventos que disparan durante la interacción del usuario con la interfaz gráfica.

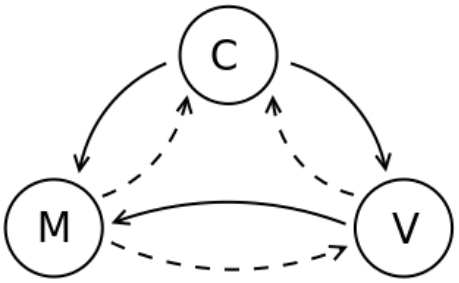

FIGURA 7.4: Comunicación en el patrón MVC.

La separación en tres partes que plantea el patrón MVC clásico es siempre útil durante el desarrollo de una aplicación, ya que nos permite mantener el código ordenado. Logra separar las distintas responsabilidades en diferentes clases, y obtener un importante grado de flexibilidad en el sistema. Durante la ejecución, los objetos vistas y sus objetos controladores pasan a componerse juntos para formar la presentación. Por cada interfaz casi siempre tendremos un par único de objetos vista-controlador acoplados.

7.6. Variaciones

Cada patrón describe un problema que ocurre una y otra vez en nuestro contexto, así como la solución a ese problema, de tal modo que se pueda aplicar esta solución un millón de veces, sin hacer lo mismo dos veces. [AIS+77]

Un patrón no se aplica a ciegas, no es una receta. Como cada problema se repite una y otra vez en contextos diferentes y únicos, el aplicar un patrón resulta en una solución también única.

Todas las aproximaciones a los patrones que se mostró pueden ser encaradas de variadas formas, y aun así responder al mismo patrón. Lo que no puede cambiar cuando aplicamos un patrón es su intensión o propósito que persigue.

El ejemplo que se mostró responde a una implementación clásica del patrón MVC. Solo hace falta cambiar de contexto para que el patrón varíe.

7.7. Trabajos futuros

A partir del estudio de patrones de diseño de software, y específicamente del patrón MVC clásico es posible divisar una amplia línea de estudios a seguir:

- MVC en la web
  Estudio de la plataforma web y aplicación del patrón.

- Frameworks MVC para desarrollo web
  Los frameworks para desarrollo web hoy en día van mucho más allá, un complejo mecanismo se esconde para simplificar el desarrollo. Los frameworks actuales como Rails, Spring MVC, Grails, Zend para distintas plataformas.

- Tests unitarios y de integración
  El patrón MVC, gracias a la separación que realiza, permite un testeo más adecuado. Técnicas ágiles como TDD, o BDD, para desarrollar sistemas a partir de test o especificaciones.

- HMVC
  Hierarchical Model View Controller es una evolución de MVC y está siendo usada ampliamente. Aparece como respuesta a problemas de escalabilidad.

- Arquitectura DCI
  Una nueva visión de la Programación Orientada a Objetos. DCI o Data Context Interaction plantea una forma adecuada para capturar el modelo cognitivo del usuario final sobre los roles y sus interacciones.

Apéndice A

Código fuente

Aquí se muestra la implementación completa del ejemplo MVC sobre la aplicación para llevar la cuenta de alumnos inscriptos en el departamento de Matemáticas. Es necesario incorporar la librería `jfreechart.jar`[1] que requiere de la librería `jcommon.jar`.

```
package mvc;

import java.awt.GridLayout;
import java.awt.ItemSelectable;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import mvc.modelo.*;
import mvc.vista.*;

public class Aplicacion {
    private DptoMatematicas modelo;
    private JFrame ventana;
    private JPanel panel;
    private JCheckBox torta, barra, alumnosLic, tabla;
    public Aplicacion() {
        modelo = new DptoMatematicas(4, 1);
        setLayout();
    }
    protected void setLayout() {
        ventana = new JFrame("Aplicacion MVC");
        ventana.setSize(300, 200);
        torta = new JCheckBox("Torta");
        barra = new JCheckBox("Barra");
        alumnosLic = new JCheckBox("Alumnos Lic.");
        tabla = new JCheckBox("Tabla");
        Listener listener = new Listener();
```

[1] http://www.jfree.org/jfreechart/download.html.

```
        torta.addItemListener(listener);
        barra.addItemListener(listener);
        alumnosLic.addItemListener(listener);
        tabla.addItemListener(listener);
        ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        panel = new JPanel();
        panel.setLayout(new GridLayout(4, 1));
        panel.add(torta);
        panel.add(barra);
        panel.add(alumnosLic);
        panel.add(tabla);
        ventana.add(panel);
        ventana.setLocation(400, 400);
        ventana.setVisible(true);
    }

    class Listener implements ItemListener {
        private Vista vistaBarra, vistaTorta, vistaSimple, vistaTabla;
        public void itemStateChanged(ItemEvent e) {
            System.out.println(e);
            if (e.getItemSelectable() == barra) {
                if (e.getStateChange() == ItemEvent.SELECTED) {
                    vistaBarra = new VistaBarra(modelo);
                } else {
                    vistaBarra.terminar();
                }
            } else if (e.getItemSelectable() == torta) {
                if (e.getStateChange() == ItemEvent.SELECTED) {
                    vistaTorta = new VistaTorta(modelo);
                } else {
                    vistaTorta.terminar();
                }
            } else if (e.getItemSelectable() == alumnosLic) {
                if (e.getStateChange() == ItemEvent.SELECTED) {
                    vistaSimple = new VistaLicenciatura(modelo);
                } else {
                    vistaSimple.terminar();
                }
            } else if (e.getItemSelectable() == tabla) {
                if (e.getStateChange() == ItemEvent.SELECTED) {
                    vistaTabla = new VistaTabla(modelo);
                } else {
                    vistaTabla.terminar();
                }
            }
        }
        private Vista crearVista(ItemSelectable itemSelectable) {
            Vista v = null;
            if (itemSelectable == barra) {
                v = new VistaTorta(modelo);
            } else if (itemSelectable == torta) {
                v = new VistaBarra(modelo);
            } else if (itemSelectable == tabla) {
                v = new VistaTabla(modelo);
            } else if (itemSelectable == alumnosLic) {
```

```
                v = new VistaLicenciatura(modelo);
            }
            return v;
        }
    }
}
```

Código A.1: Archivo: `/mvc/Aplicacion.java`.

```
package mv;
public interface Observador {
    public abstract void actualizar();
}
```

Código A.2: Archivo: `/mvc/Observador.java`.

```
package mvc;
import java.util.ArrayList;
public abstract class Sujeto {
    private ArrayList<Observador> observadores = new ArrayList<Observador>();
    public void registrar(Observador o) {
        observadores.add(o);
    }
    public void quitar(Observador o) {
        observadores.remove(o);
    }
    public void notificarObservadores() {
        for (Observador o : observadores) {
            o.actualizar();
        }
    }
}
```

Código A.3: Archivo: `/mvc/Sujeto.java`.

```
package mvc;
public class Test {
    public static void main(String[] arg) {
        Aplicacion aplicacion = new Aplicacion();
    }
}
```

Código A.4: Archivo: `/mvc/Test.java`.

```
package mvc.controlador;
import mvc.*;
import mvc.modelo.*;
import mvc.vista.*;
public abstract class Controlador implements Observador {
```

```
    private Modelo modelo;
    private Vista vista;
    public Controlador(Modelo modelo, Vista vista) {
        this.vista = vista;
        this.modelo = modelo;
        modelo.registrar(this);
        actualizar();
    }
    public void actualizar() {
    }
    protected Modelo getModelo() {
        return modelo;
    }
    protected Vista getVista() {
        return vista;
    }
    public void terminar() {
        modelo.quitar(this);
        modelo = null;
        vista = null;
    }
}
```

Código A.5: Archivo: `/mvc/controlador/Controlador.java`.

```
package mvc.controlador;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
import mvc.modelo.*;
import mvc.vista.*;
public class ControladorLicenciatura extends Controlador {
    public ControladorLicenciatura(Modelo modelo, Vista vista) {
        super(modelo, vista);
        VistaLicenciatura vl = (VistaLicenciatura) vista;
        vl.addListener(new ControladorAlumnos());
    }
    private class ControladorAlumnos implements ChangeListener {
        public void stateChanged(ChangeEvent e) {
            VistaLicenciatura vl = (VistaLicenciatura) getVista();
            getModelo().setAlumnosLic(vl.getAlumnosLicenciatura());
        }
    }
}
```

Código A.6: Archivo: `/mvc/controlador/ControladorLicenciatura.java`.

```
package mvc.controlador;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import mvc.modelo.*;
import mvc.vista.*;
public class ControladorTabla extends Controlador {
```

```
    public ControladorTabla(Modelo modelo, Vista vista) {
        super(modelo, vista);
        VistaTabla vistaTabla = (VistaTabla) vista;
        vistaTabla.addAlumnosLicListener(new AlumnosLicControlador());
        vistaTabla.addAlumnosProfListener(new AlumnosProfControlador());
    }
    private class AlumnosLicControlador implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            VistaTabla vt = (VistaTabla) getVista();
            getModelo().setAlumnosLic(Integer.parseInt(vt.getAlumnosLic()));
        }
    }
    private class AlumnosProfControlador implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            VistaTabla vt = (VistaTabla) getVista();
            getModelo().setAlumnosProf(Integer.parseInt(vt.getAlumnosProf()));
        }
    }
}
```

Código A.7: Archivo: `/mvc/controlador/ControladorTabla.java`.

```
package mvc.modelo;
import mvc.Sujeto;
public class DptoMatematicas extends Sujeto implements Modelo {
    private int alumnosLic;
    private int alumnosProf;
    public DptoMatematicas(int alumnosLic, int alumnosProf) {
        this.alumnosLic = alumnosLic;
        this.alumnosProf = alumnosProf;
    }
    public void setAlumnosLic(int alumnos) {
        if (validar(alumnos) && alumnos != this.alumnosLic) {
            this.alumnosLic = alumnos;
            notificarObservadores();
        }
    }
    public void setAlumnosProf(int alumnos) {
        if (validar(alumnos) && alumnos != this.alumnosProf) {
            this.alumnosProf = alumnos;
            notificarObservadores();
        }
    }
    public int getAlumnosLic() {
        return alumnosLic;
    }
    public int getAlumnosProf() {
        return alumnosProf;
    }
    private boolean validar(int dato) {
        return (dato >= 0);
    }
```

```
}
```

Código A.8: Archivo: `/mvc/modelo/DptoMatematicas.java`.

```
package mvc.modelo;
import mvc.Observador;
public interface Modelo {
    public void setAlumnosLic(int alumnos);
    public void setAlumnosProf(int alumnos);
    public int getAlumnosLic();
    public int getAlumnosProf();
    public void registrar(Observador o);
    public void quitar(Observador o);
}
```

Código A.9: Archivo: `/mvc/modelo/Modelo.java`.

```
package mvc.vista;
import java.awt.Component;
import java.awt.Container;
import javax.swing.AbstractButton;
import javax.swing.JFrame;
import mvc.*;
import mvc.controlador.*;
import mvc.modelo.*;
public abstract class Vista implements Observador {
    private Modelo modelo;
    private Controlador controlador;
    JFrame ventana;
    public Vista(Modelo modelo) {
        JFrame.setDefaultLookAndFeelDecorated(true);
        ventana = new JFrame();
        removeMinMaxClose(ventana);
        ventana.setResizable(false);
        this.modelo = modelo;
        modelo.registrar(this);
        this.setLayout();
        this.controlador = fabricarControlador();
        this.actualizar();
    }
    protected abstract void setLayout();
    protected Controlador fabricarControlador() {
        return null;
    }
    public Controlador getControlador() {
        return controlador;
    }
    public Modelo getModelo() {
        return modelo;
    }
    public void terminar() {
        if (controlador != null) {
```

```
            controlador.terminar();
            controlador = null;
        }
        modelo.quitar(this);
        modelo = null;
        ventana.dispose();
    }
    public void removeMinMaxClose(Component comp) {
        if (comp instanceof AbstractButton) {
            comp.getParent().remove(comp);
        }
        if (comp instanceof Container) {
            Component[] comps = ((Container) comp).getComponents();
            for (int x = 0, y = comps.length; x < y; x++) {
                removeMinMaxClose(comps[x]);
            }
        }
    }
}
```

Código A.10: Archivo: `/mvc/vista/Vista.java`.

```
package mvc.vista;
import javax.swing.JFrame;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.category.DefaultCategoryDataset;
import mvc.modelo.*;
public class VistaBarra extends Vista {
    private DefaultCategoryDataset dataset;
    private JFreeChart jfc;
    public VistaBarra(Modelo modelo) {
        super(modelo);
    }
    @Override
    protected void setLayout() {
        dataset = new DefaultCategoryDataset();
        ventana.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        ventana.setTitle("Departamento Matematicas");
        ventana.setSize(300, 200);
        jfc = ChartFactory.createBarChart("", "", "", dataset, PlotOrientation.
    VERTICAL, true, false, false);
        ventana.setContentPane(new ChartPanel(jfc));
        ventana.setLocation(1000, 100);
        ventana.setVisible(true);
    }
    @Override
    public void actualizar() {
        dataset.setValue(new Double(getModelo().getAlumnosLic()), "Lic.", "
    Alumnos");
        dataset.setValue(new Double(getModelo().getAlumnosProf()), "Prof.", "
    Alumnos");
```

```
    }
}
```

Código A.11: Archivo: /mvc/vista/VistaBarra.java.

```
package mvc.vista;
import java.awt.GridLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSpinner;
import javax.swing.event.ChangeListener;
import mvc.controlador.*;
import mvc.modelo.*;
public class VistaLicenciatura extends Vista {
    private JLabel etiquetaLic;
    private JSpinner spinnerLic;
    private JPanel panel;
    public VistaLicenciatura(Modelo modelo) {
        super(modelo);
    }
    @Override
    protected void setLayout() {
        ventana.setTitle("Licenciatura Matematicas");
        ventana.setSize(300, 70);
        ventana.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        panel = new JPanel();
        panel.setLayout(new GridLayout(1, 2));
        etiquetaLic = new JLabel("Alumnos");
        etiquetaLic.setHorizontalAlignment(JLabel.CENTER);
        spinnerLic = new JSpinner();
        panel.add(etiquetaLic);
        panel.add(spinnerLic);
        ventana.add(panel);
        ventana.setLocation(100, 100);
        ventana.setVisible(true);
    }
    @Override
    protected Controlador fabricarControlador() {
        return new ControladorLicenciatura(getModelo(), this);
    }
    public int getAlumnosLicenciatura() {
        return (Integer) spinnerLic.getModel().getValue();
    }
    @Override
    public void actualizar() {
        spinnerLic.setValue(getModelo().getAlumnosLic());
    }
    public void addListener(ChangeListener controller) {
        spinnerLic.addChangeListener(controller);
    }
}
```

Código A.12: Archivo: /mvc/vista/VistaLicenciatura.java.

```
package mvc.vista;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import javax.swing.*;
import mvc.controlador.*;
import mvc.modelo.*;
public class VistaTabla extends Vista {
    private JTextField alumnosLic, alumnosProf;
    private JLabel etiquetaLic, etiquetaProf;
    private JPanel panel;
    public VistaTabla(Modelo modelo) {
        super(modelo);
    }
    @Override
    protected Controlador fabricarControlador() {
        return new ControladorTabla(getModelo(), this);
    }
    @Override
    public void actualizar() {
        alumnosLic.setText(Integer.toString(getModelo().getAlumnosLic()));
        alumnosProf.setText(Integer.toString(getModelo().getAlumnosProf()));
    }
    @Override
    protected void setLayout() {
        ventana.setTitle("Departamento Matematicas");
        ventana.setSize(300, 200);
        ventana.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        panel = new JPanel();
        panel.setLayout(new GridLayout(3, 2));
        alumnosLic = new JTextField();
        alumnosProf = new JTextField();
        alumnosLic.setActionCommand("Licenciatura");
        alumnosProf.setActionCommand("Profesorado");
        etiquetaLic = new JLabel("Alumnos Lic.");
        etiquetaProf = new JLabel("Alumnos Prof.");
        etiquetaLic.setHorizontalAlignment(JLabel.CENTER);
        etiquetaProf.setHorizontalAlignment(JLabel.CENTER);
        panel.add(etiquetaLic);
        panel.add(alumnosLic);
        panel.add(etiquetaProf);
        panel.add(alumnosProf);
        ventana.add(panel);
        ventana.setLocation(400, 100);
        ventana.setVisible(true);
    }
    public String getAlumnosLic() {
        return alumnosLic.getText();
    }
    public String getAlumnosProf() {
        return alumnosProf.getText();
    }
    public void addAlumnosLicListener(ActionListener controller) {
        alumnosLic.addActionListener(controller);
    }
    public void addAlumnosProfListener(ActionListener controller) {
```

```
        alumnosProf.addActionListener(controller);
    }
}
```

Código A.13: Archivo: /mvc/vista/VistaTabla.java.

```
package mvc.vista;
import javax.swing.JFrame;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.data.general.DefaultPieDataset;
import mvc.modelo.*;
public class VistaTorta extends Vista {
    private DefaultPieDataset dataset;
    private JFreeChart jfc;
    public VistaTorta(DptoMatematicas modelo) {
        super(modelo);
    }
    @Override
    protected void setLayout() {
        dataset = new DefaultPieDataset();
        ventana.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        ventana.setTitle("Departamento Matematicas");
        ventana.setSize(300, 200);
        jfc = ChartFactory.createPieChart("Alumnos", dataset, true, true, false);
        ventana.setContentPane(new ChartPanel(jfc));
        ventana.setLocation(700, 100);
        ventana.setVisible(true);
    }
    @Override
    public void actualizar() {
        dataset.setValue("Lic.", getModelo().getAlumnosLic());
        dataset.setValue("Prof.", getModelo().getAlumnosProf());
    }
}
```

Código A.14: Archivo: /mvc/vista/VistaTorta.java.

Apéndice B

Línea de tiempo

En la página siguiente se muestra una línea de tiempo situando los eventos y personalidades que se nombran en la tesis, y otros que no se nombran pero también son de importancia para situar al lector en el contexto histórico relacionado.

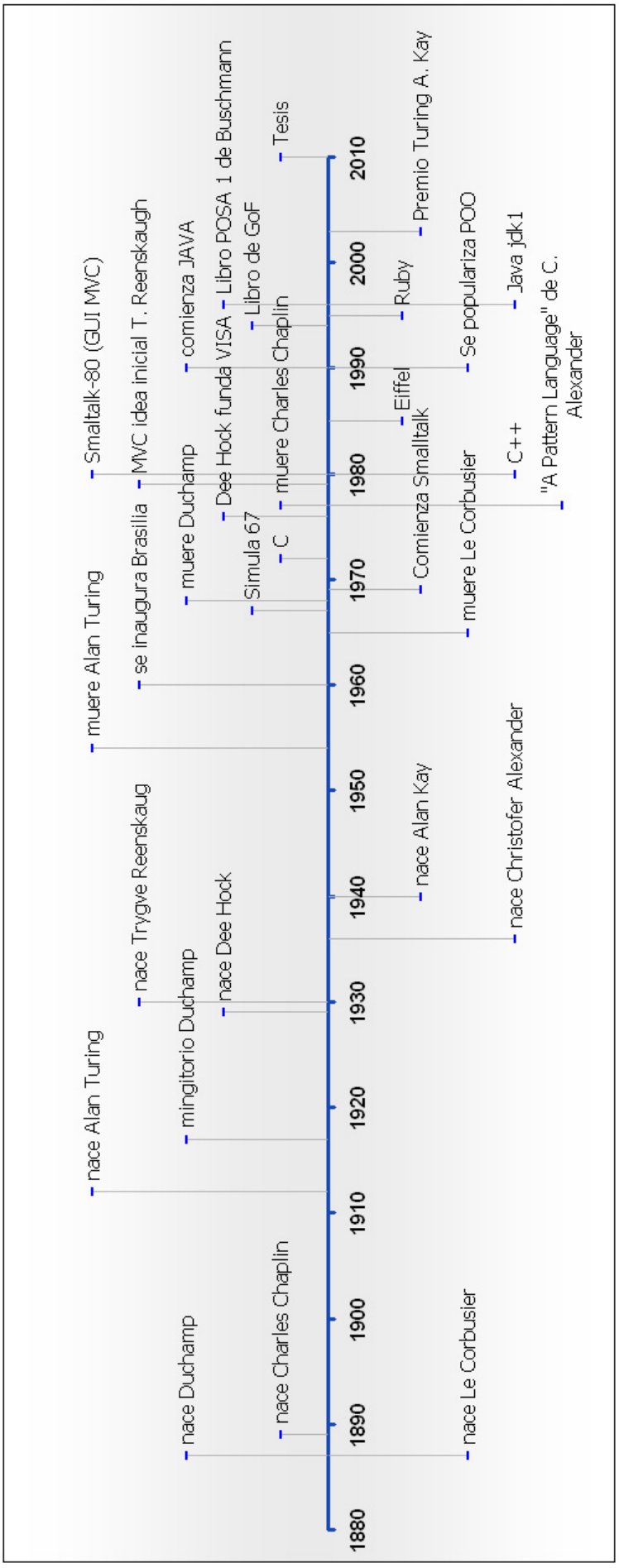

Figura B.1: Línea de tiempo de eventos importantes.

Bibliografía

[Abe09] Samisa Abeysinghe. *PHP Team Development*. Pack Publishing, September 2009.

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, 1977.

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[Bec07] Kent Beck. *Implementation Patterns*. Addison-Wesley Professional, 1st edition, November 2007.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1st edition, August 1996.

[C2W] Cunningham & cunningham, inc. *Portland Pattern Repository's Wiki*. Recurso disponible online: http://www.c2.com/.

[CBB+02] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison Wesley, September 2002.

[FFBS04] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, 1st edition, October 2004.

[Fow01] Martin Fowler. Separating user interface code. *IEEE Software magazine*, 18(2):96–97, mar–apr 2001.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.

[Gal07] Wilbert O. Galitz. *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*. Wiley Publishing, Inc., 3rd edition, 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition, January 1995.

[Gor06] Ian Gorton. *Essencial Software Architecture*. Springer, 2006.

[HBC+96] Hewett, Baecker, Card, Carey, Gasen, Mantei, Perlman, Strong, and Verplank. *ACM SIGCHI Curricula for Human-Computer Interaction*. 1996. http://old.sigchi.org/cdg/index.html.

[PM02] Richard Pawson and Robert Matthews. *Naked Objects*. Wiley, 1st edition, November 2002. Online edition: http://www.nakedobjects.org/book/.

[Ree] Trygve M. H. Reenskaug. Pages of trygve m. h. reenskaug. Recurso online: http://heim.ifi.uio.no/~trygver/index.html.

[Ree79a] Trygve Reenskaug. "models-views-controllers". *Technical note at Xerox PARC*, dec 1979. A scanned version on http://heim.ifi.uio.no/~trygver/mvc/index.html.

[Ree79b] Trygve Reenskaug. Thing-model-view-editor an example from a planning system. *Technical note at Xerox PARC*, May 1979. A scanned version on http://heim.ifi.uio.no/~trygver/mvc/index.html.

[Ree03] Trygve Reenskaug. The model-view-controller (mvc) its past and present. *Abstract, University of Oslo, 2003*, 2003.

[RM06] Martin C. Robert and Martin Micah. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, July 2006.

[WIK] Wikipedia. Recurso online: http://www.wikipedia.org/.