|  |
| --- |
| **Práctica Final Junio** |
| Examen Final 2013 |
|  |
| **Juan Carlos Marco González, Juan Luis Pérez Valbuena, Emilio Álvarez Piñeiro.** |
| **14/06/2013** |
|  |
|  |
|  |

Contenido

[Preliminares 3](#__RefHeading__250_860113062)

[Requisitos: 3](#__RefHeading__334_1925510426)

[Objetivos 3](#__RefHeading__336_1925510426)

[Introducción 5](#__RefHeading__338_1925510426)

[Estado del Arte 5](#__RefHeading__340_1925510426)

[Fundamentación teórica: 5](#__RefHeading__342_1925510426)

[Propuestas de Solución (Solución implementada) 5](#__RefHeading__344_1925510426)

[Discusión 5](#__RefHeading__346_1925510426)

[Conclusión 5](#__RefHeading__348_1925510426)

[Líneas futuras 6](#__RefHeading__350_1925510426)

[Referencias bibliográficas utilizadas 7](#__RefHeading__352_1925510426)

[Anexos 7](#__RefHeading__354_1925510426)

# Preliminares

**Título:** Examen Final Junio

**Fecha de entrega:** 21 de Junio de 2013.

## Requisitos:

* Conocer toda la teoría dada en la cátedra
* Tener claros los conceptos vistos en prácticas anteriores.
* Tener claros los conceptos para el uso de teclado y ﬁcheros en java y su problemática de uso con swing.
* Profundizar en los conceptos sobre polimorﬁsmo, genéricos, etc.
* Conocimientos básicos de ciclismo.(Se aportan por la cátedra)

## Objetivos

Demostrar que los conocimientos y aptitudes adquiridas durante el desarrollo del curso académico son suficientes para realizar una aplicación que se ajuste a los factores de calidad del Software

### General:

* Aﬁanzar los conocimientos adquiridos de Metodología de la Programación y añadir los recibidos sobre el uso de ﬁcheros, interacción con usuarios, etc...
* Incluir lo aprendido sobre genéricos a la práctica 1 como siguiente nivel de desarrollo.

### Particulares:

* Estudiar la interacción con usuarios mediante teclado y pantalla.
* Estudiar el uso de ﬁcheros, su comportamiento y compartición.
* Aplicar los conocimientos adquiridos sobre genéricos (enumerados, mapas, etc…).
* Ampliar el número de actores y objetivos de **cambios dinámicos** en el programa para orientar la práctica hacia el siguiente nivel, que se corresponderá con la práctica 3.
* Aﬁanzar conceptos de Física básica, aplicados a la bicicleta, claramente enmohecidos.
* Utilizar Eclipse y JUnit.

## Resumen:

Utilizando todos los conocimientos adquiridos durante el curso académico de Tecnología de Programación se ha realizado una aplicación basada en una carrera de ciclistas. Se incluyen numerosos patrones de programación así como una salida de datos adecuada. Se ha realizado una verificación y validación exhaustiva.

## Abstract

This application is based on a cycle ride.

The user, can send commands to cyclist, and this, send messages to his bike.

For example, if you send to cyclist the command brake. The cyclist send a message to his bike and this gradually slow.

Also this application contents external factors. The user can change the wind and slope typing commands.

The synchronization in all objects, is done through for the class reloj, because this class for each second.

In this aplication, have been included programming patterns, like patterns command, singletone and mediator.

## OBservaciones

Ninguna reseñable.

## Sugerencias

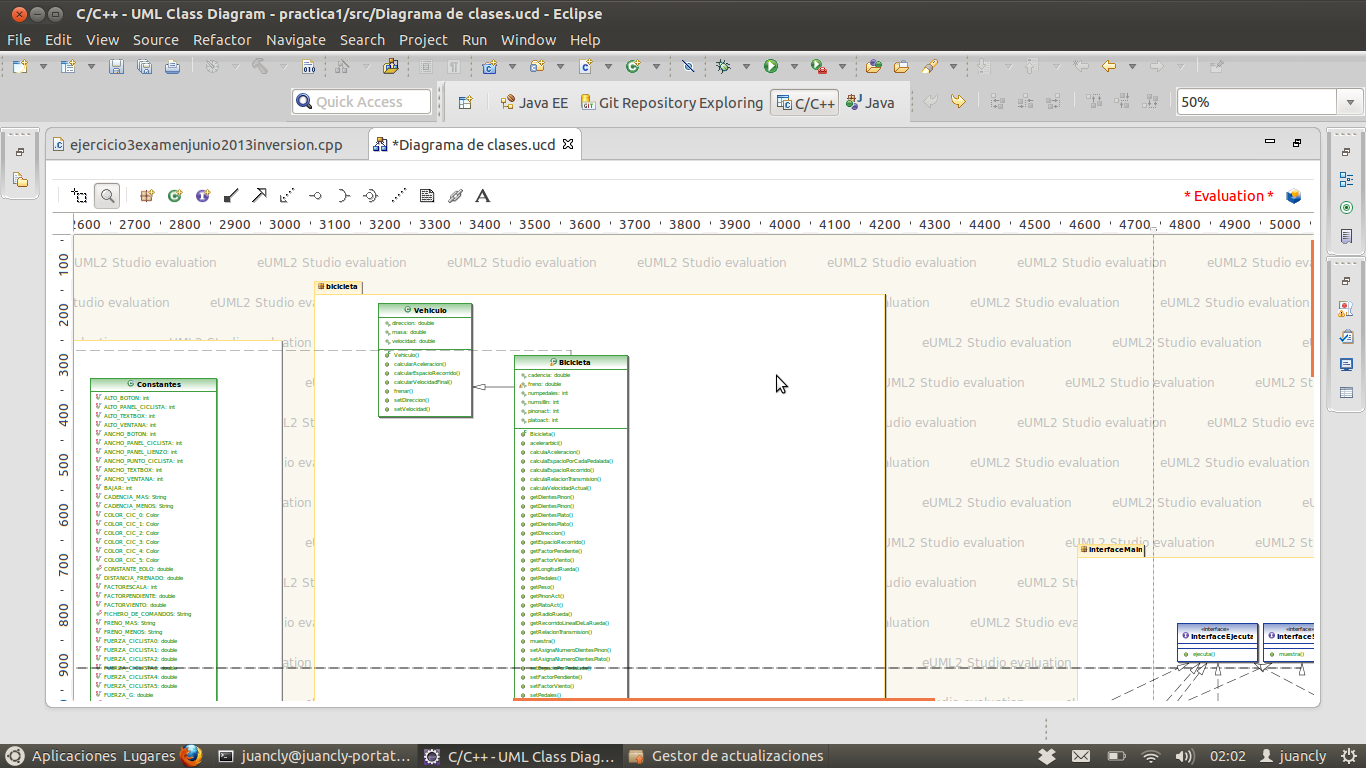
# Introducción

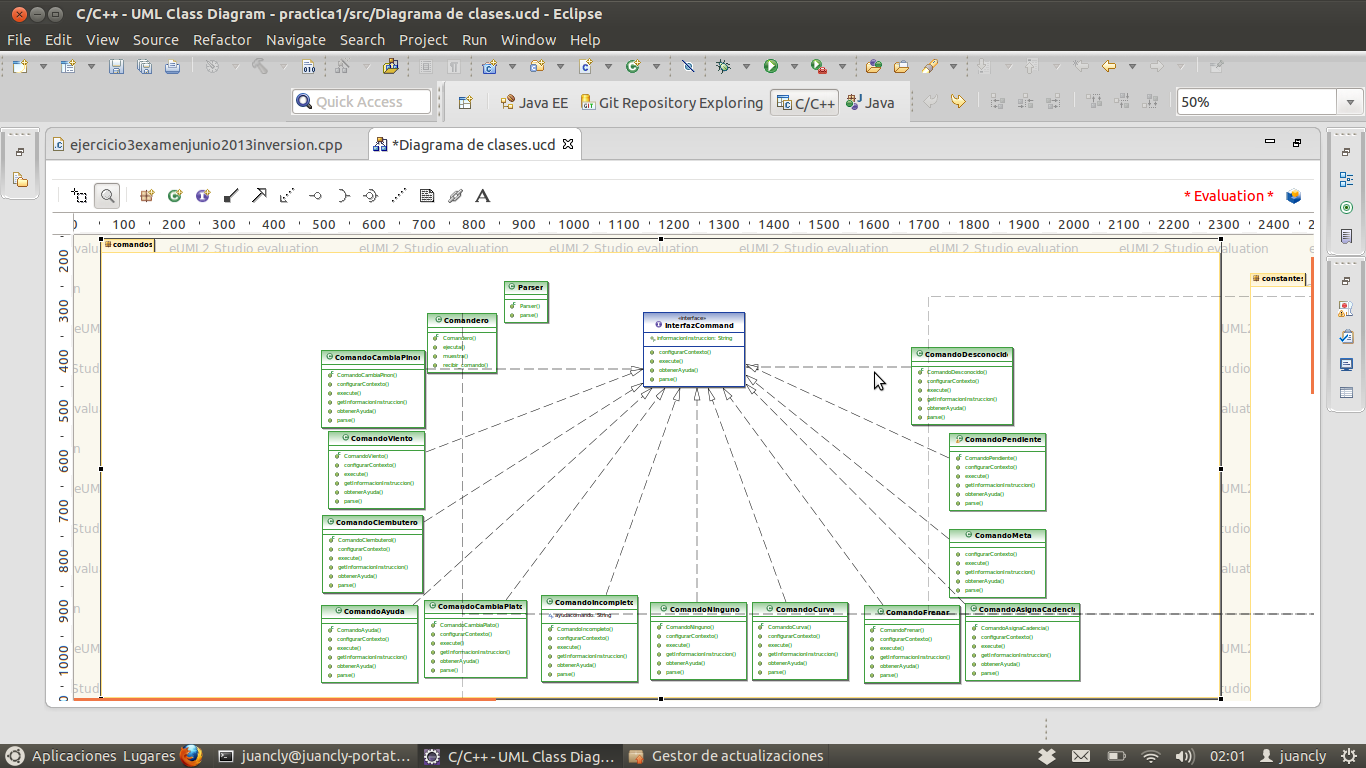
## Estado del Arte y Fundamentación teórica

Nuestra práctica se basa en el patrón Modelo Vista Controlador (MVC) y el uso de múltiples patrones de programación así como Interfaces.

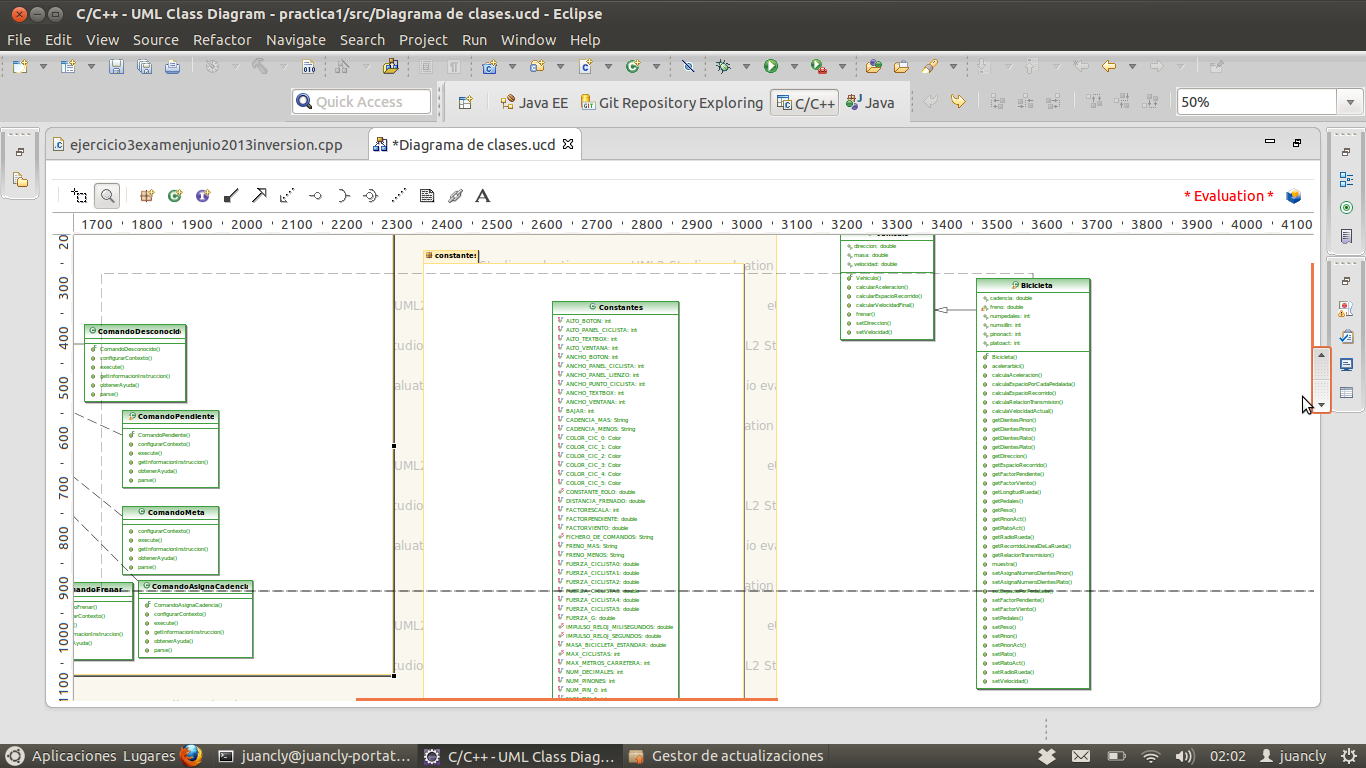
### 2.1.0 Diagrama de clases (Dividido en paquetes)

PAQUETE BICICLETA

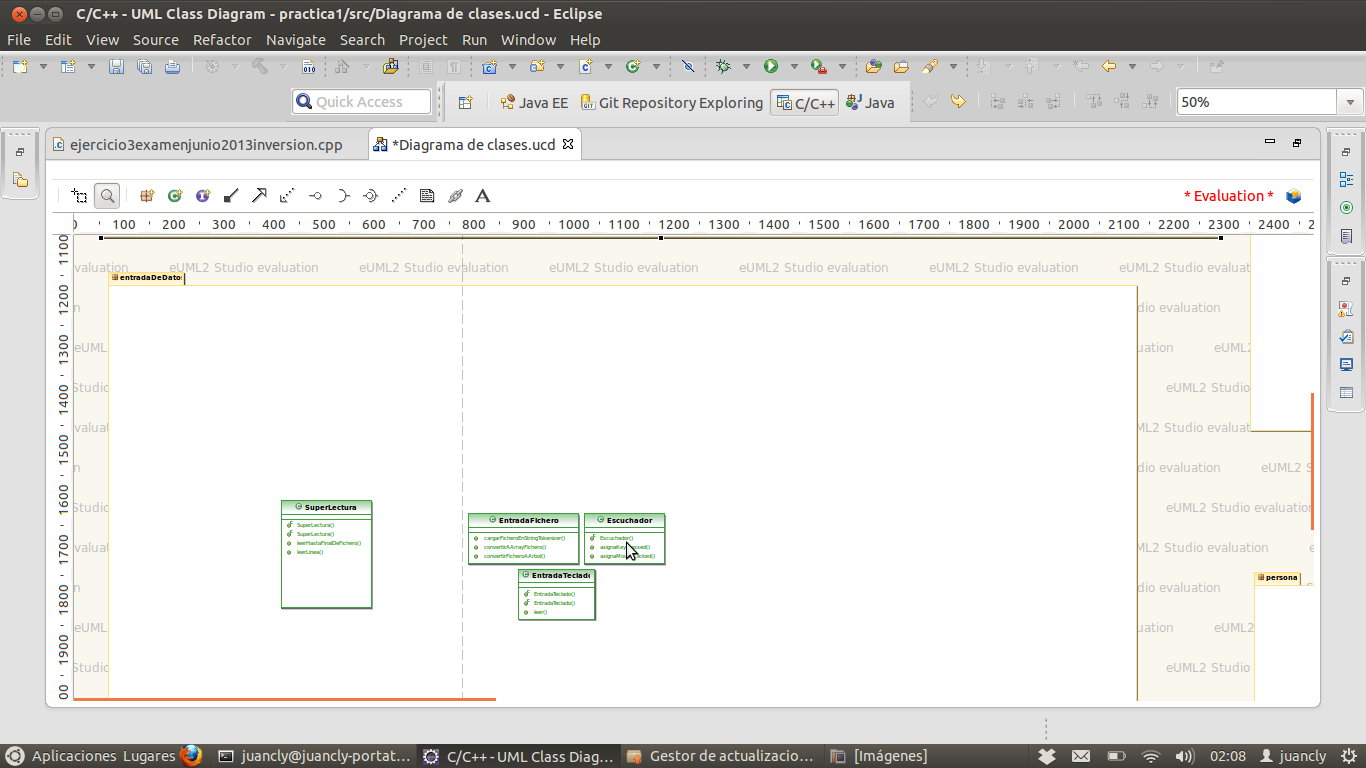


PAQUETE COMANDOS

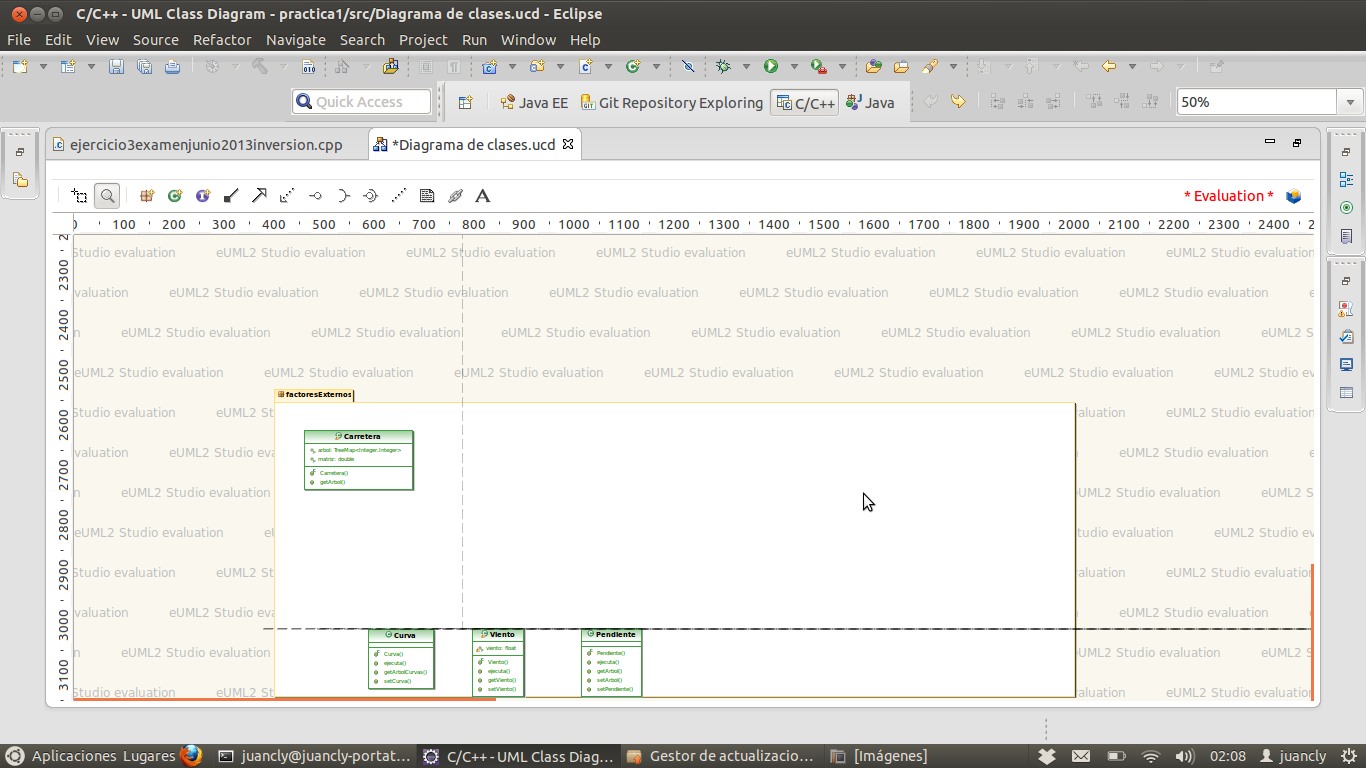
PAQUETE CONSTANTES



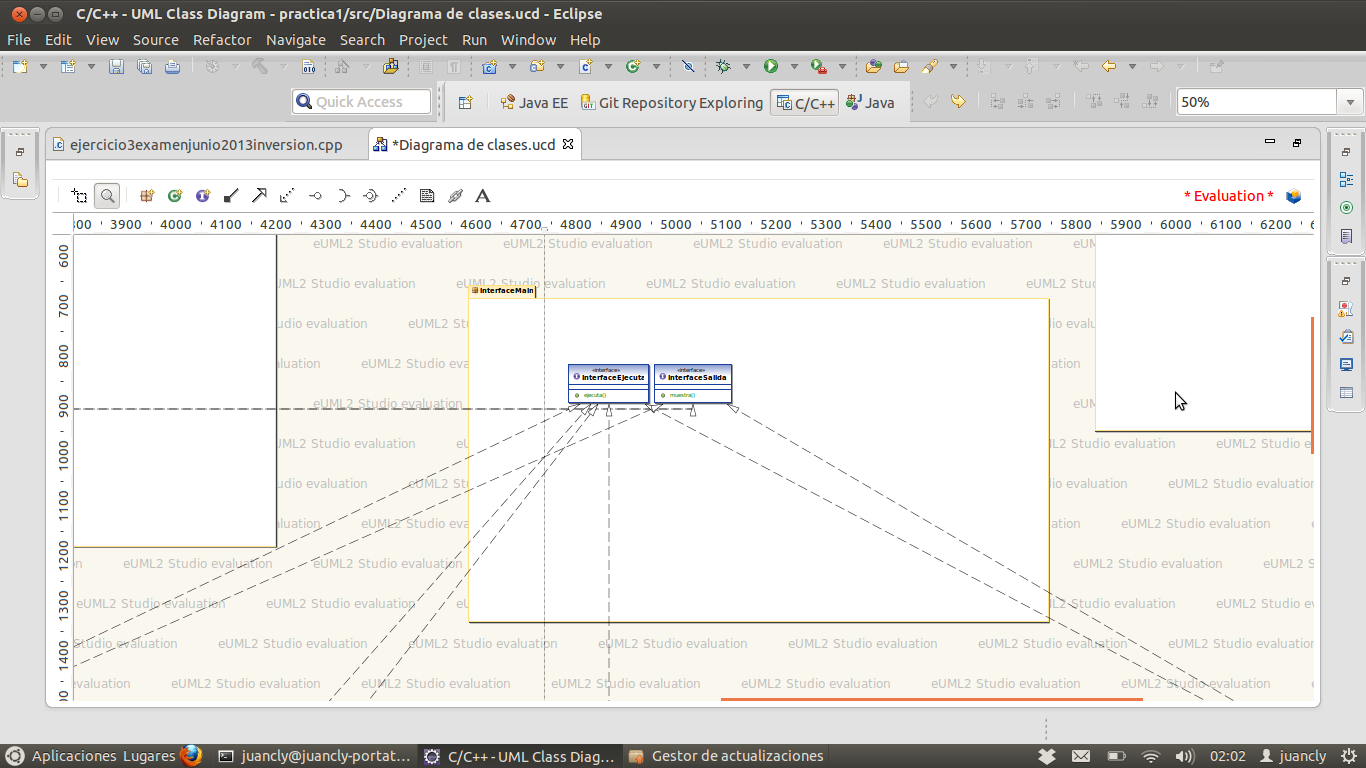
PAQUETE ENTRADA DE DATOS



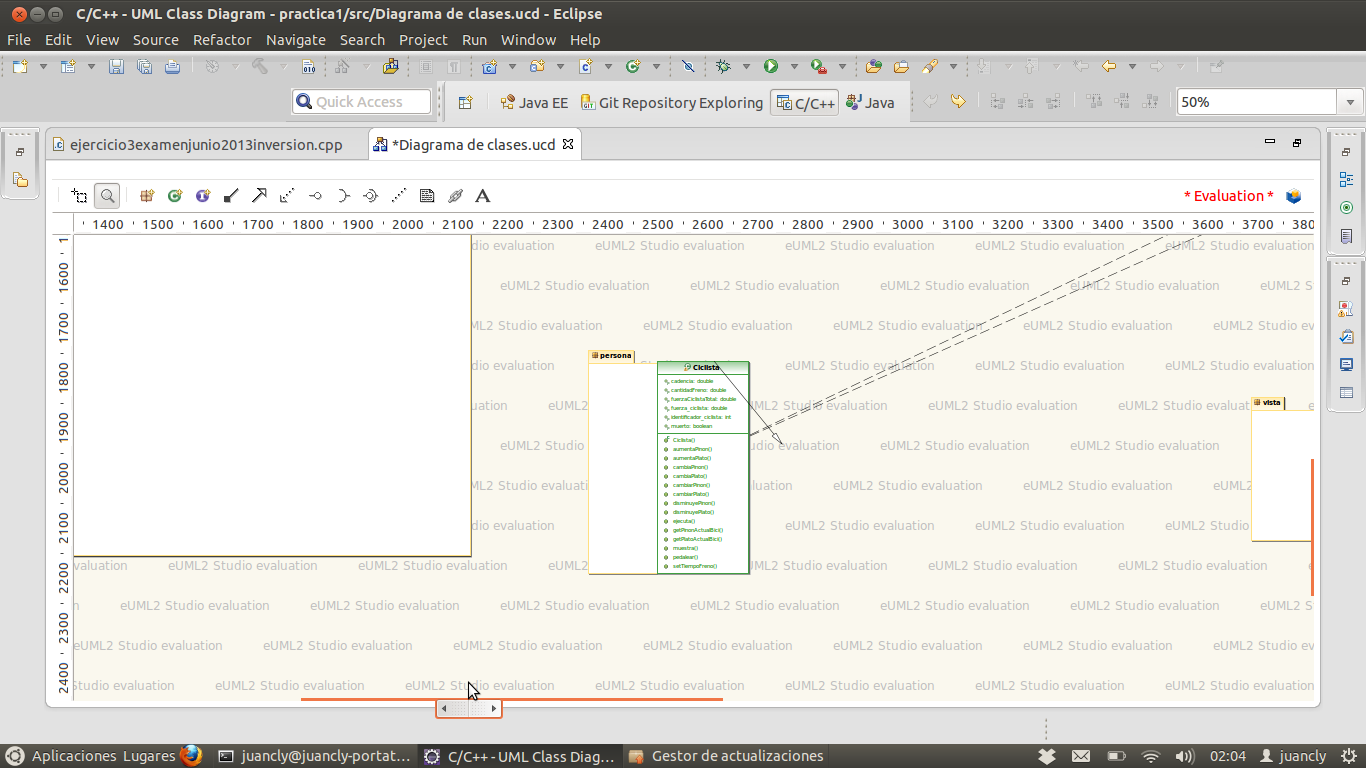
PAQUETE FACTORES EXTERNOS



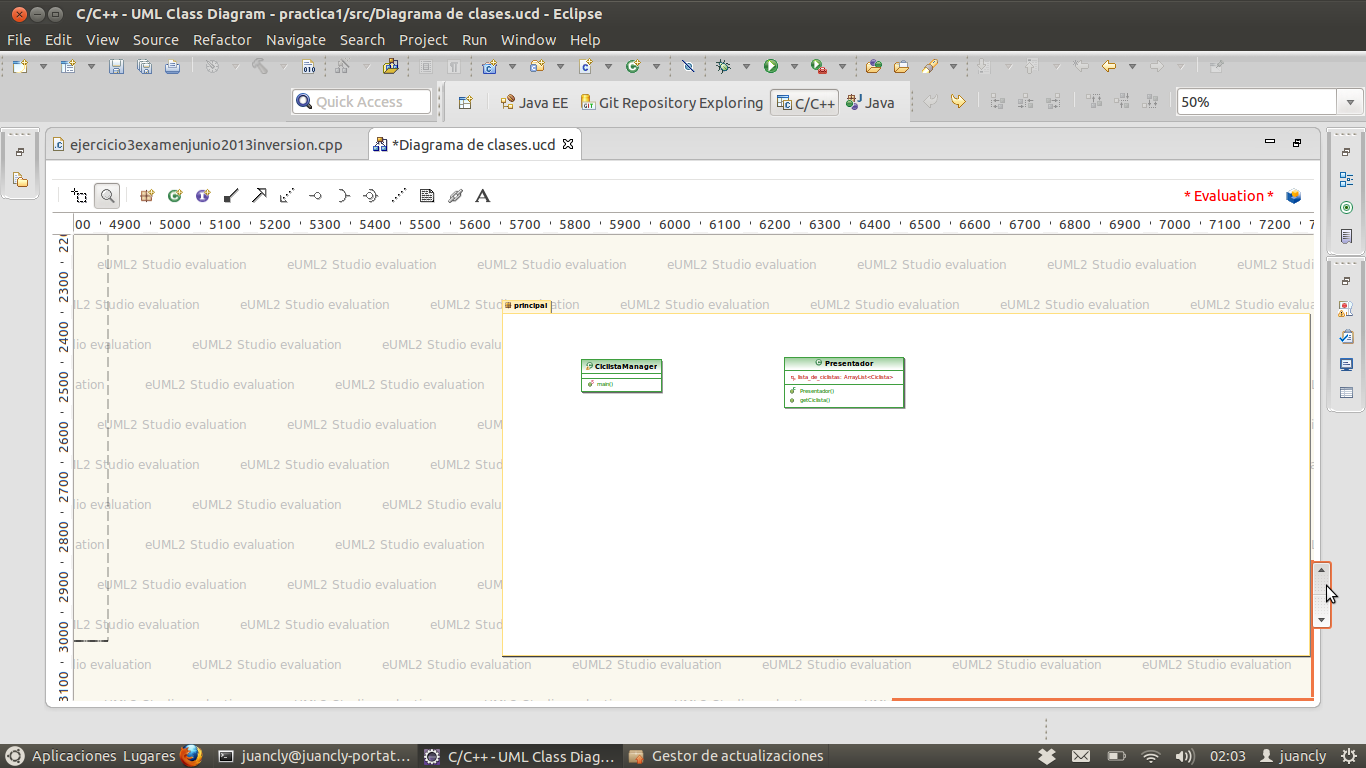
PAQUETE INTERFACEMAIN

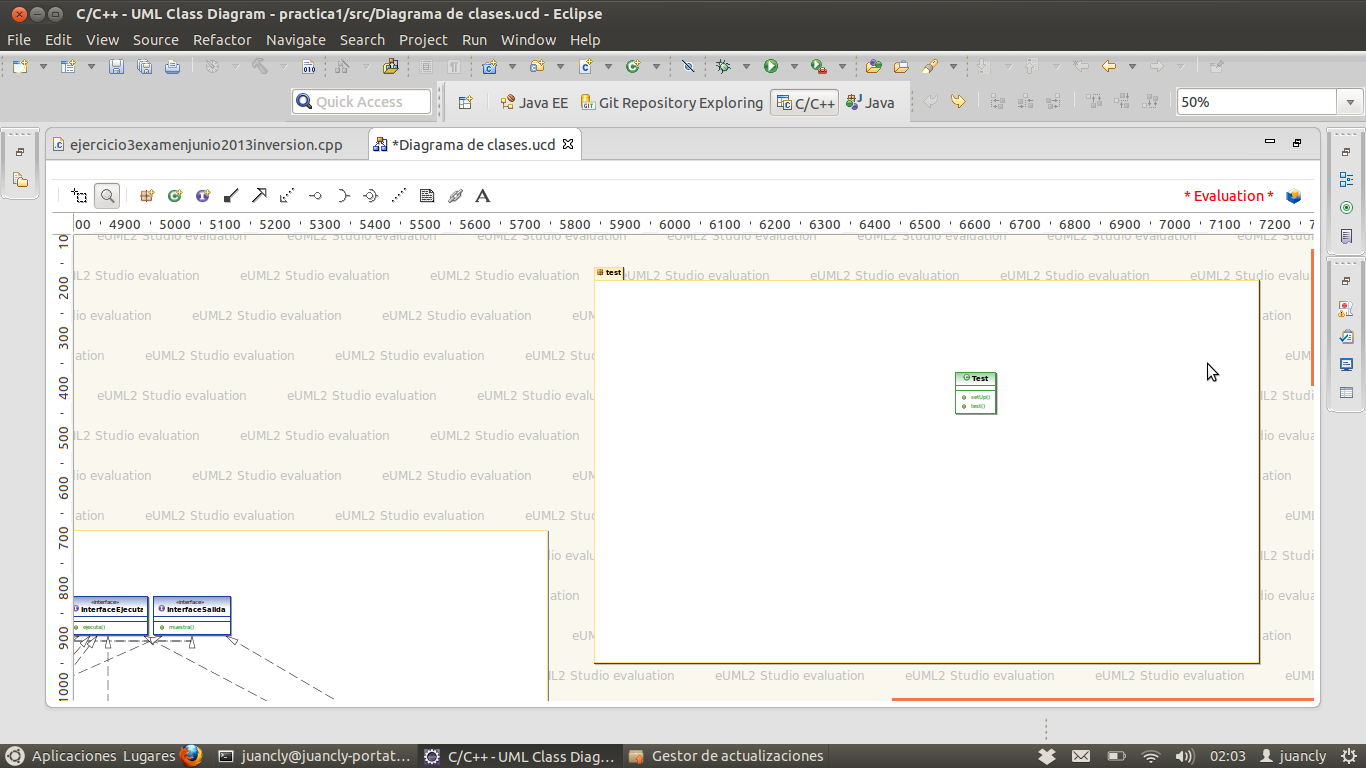


PAQUETE PERSONA

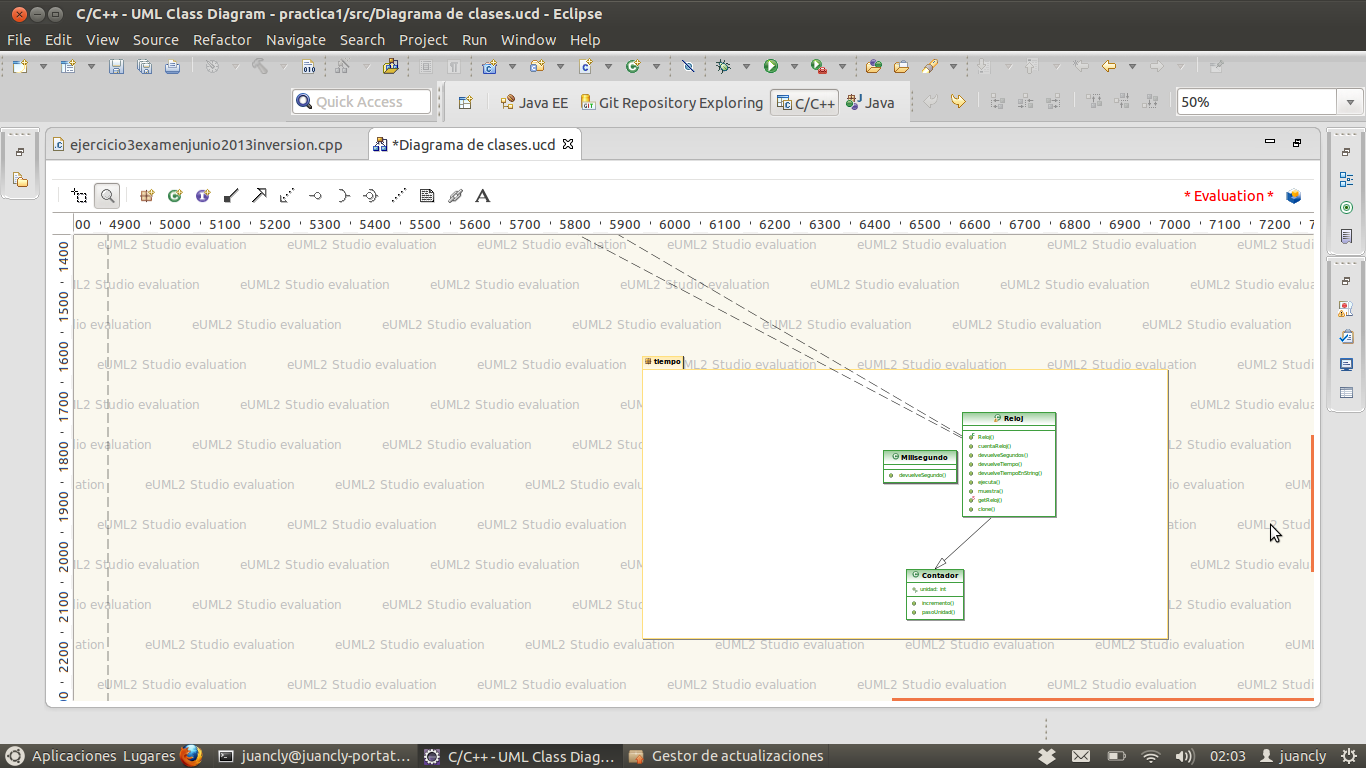


PAQUETE PRINCIPAL

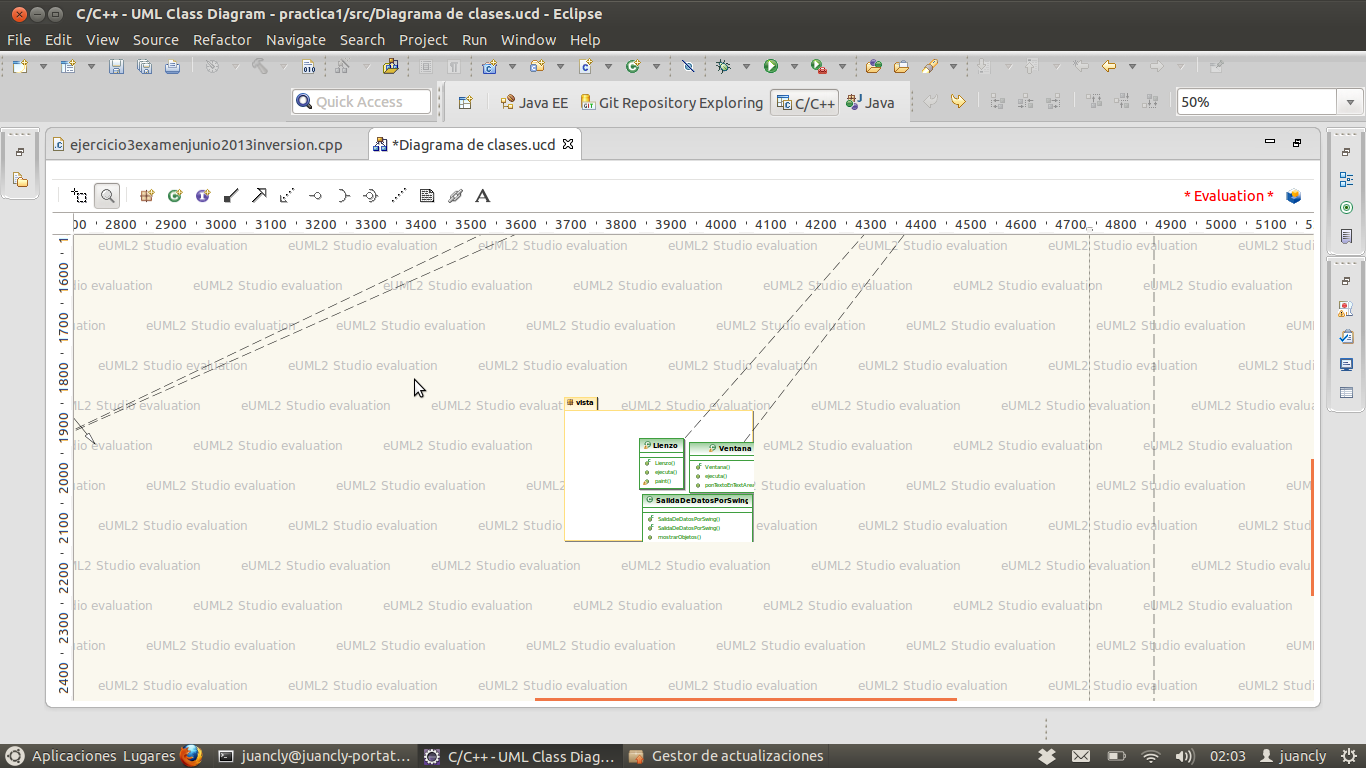


PAQUETE TEST

PAQUETE RELOJ



PAQUETE VISTA



### Modelo Vista Controlador

El **Modelo Vista Controlador (MVC)** es un patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el **modelo**, la **vista** y el **controlador**, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario. Este patrón de diseño se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

De manera genérica, los componentes de MVC se podrían definir como sigue:

* **Modelo:** Es la representación específica de la información con la cual el sistema opera, por lo tanto gestiona todos los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (lógica de negocio). Envía a la 'vista' aquella parte de la información que en cada momento se le solicita para que sea mostrada (típicamente a un usuario). Las peticiones de acceso o manipulación de información llegan al 'modelo' a través del 'controlador'12 .
* **Controlador**: Responde a eventos (usualmente acciones del usuario) e invoca peticiones al 'modelo' cuando se hace alguna solicitud sobre la información (por ejemplo, editar un documento o un registro en una base de datos). También puede enviar comandos a su 'vista' asociada si se solicita un cambio en la forma en que se presenta de 'modelo' (por ejemplo, desplazamiento o scroll por un documento o por los diferentes registros de una base de datos), por tanto se podría decir que el 'controlador' hace de intermediario entre la 'vista' y el 'modelo' (véase Middleware).
* **Vista:** Presenta el 'modelo' (información y lógica de negocio) en un formato adecuado para interactuar (usualmente la interfaz de usuario) por tanto requiere de dicho 'modelo' la información que debe representar como salida.

### Patrones de Diseño

Los **patrones de diseño** son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón de diseño resulta ser una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su **efectividad** resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser **reutilizable**, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

#### Singleton

El patrón de diseño **singleton** (instancia única) está diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto.

Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

El patrón *singleton* se implementa creando en nuestra clase un método que crea una instancia del objeto sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado).

La instrumentación del patrón puede ser delicada en programas con múltiples hilos de ejecución. Si dos hilos de ejecución intentan crear la instancia al mismo tiempo y esta no existe todavía, sólo uno de ellos debe lograr crear el objeto. La solución clásica para este problema es utilizar exclusión mutua en el método de creación de la clase que implementa el patrón.

Las situaciones más habituales de aplicación de este patrón son aquellas en las que dicha clase controla el acceso a un recurso físico único (como puede ser el ratón o un archivo abierto en modo exclusivo) o cuando cierto tipo de datos debe estar disponible para todos los demás objetos de la aplicación.

El patrón *singleton* provee una única instancia global gracias a que:

* La propia clase es responsable de crear la única instancia.
* Permite el acceso global a dicha instancia mediante un método de clase.
* Declara el constructor de clase como privado para que no sea instanciable directamente.

#### Mediator

El patrón **mediador** define un objeto que encapsula cómo un conjunto de objetos interactúan. Este patrón de diseño está considerado como un patrón de comportamiento debido al hecho de que puede alterar el comportamiento del programa en ejecución.

Habitualmente un programa está compuesto de un número de clases (muchas veces elevado). La lógica y computación es distribuída entre esas clases. Sin embargo, cuantas más clases son desarrolladas en un programa, especialmente durante mantenimiento y/o refactorización, el problema de comunicación entre estas clases quizás llegue a ser más complejo. Esto hace que el programa sea más difícil de leer y mantener. Además, puede llegar a ser difícil cambiar el programa, ya que cualquier cambio podría afectar código en muchas otras clases.

Con el patrón mediador, la comunicación entre objetos es encapsulada con un objeto mediador. Los objetos no se comunican de forma directa entre ellos, en lugar de ello se comunican mediante el mediador. Esto reduce las dependencias entre los objetos en comunicación, reduciendo entonces la Dependencia de código.

#### Command

El patrón **command** permite solicitar una operación a un objeto sin conocer realmente el contenido de esta operación, ni el receptor real de la misma. Para ello se encapsula la petición como un objeto, con lo que además se facilita la parametrización de los métodos.

* Encapsula un mensaje como un objeto, con lo que permite gestionar colas o registro de mensaje y deshacer operaciones.
* Soportar restaurar el estado a partir de un momento dado.
* Ofrecer una interfaz común que permita invocar las acciones de forma uniforme y extender el sistema con nuevas acciones de forma más sencilla.

El concepto de "orden" puede ser ambiguo y complejo en los sistemas actuales y al mismo tiempo muy extendido: intérpretes de órdenes del sistema operativo, lenguajes de macros de paquetes ofimáticos, gestores de bases de datos, protocolos de servidores de Internet, etc.

Este patrón presenta una forma sencilla y versátil de implementar un sistema basado en comandos facilitándose su uso y ampliación.

* Facilitar la parametrización de las acciones a realizar.
* Independizar el momento de petición del de ejecución.
* Implementar CallBacks, especificando que órdenes queremos que se ejecuten en ciertas situaciones de otras órdenes. Es decir, un parámetro de una orden puede ser otra orden a ejecutar.
* Soportar el "deshacer".
* Desarrollar sistemas utilizando órdenes de alto nivel que se construyen con operaciones sencillas (primitivas).

# pROPUESTAS DE SOLUCIÓN

Como se trata de una práctica acumulativa, no se han discutido diferentes soluciones, simplemente se ha añadido las funcionalidades pertinentes. Esto demuestra la fácil escalabilidad del sistema.

Los patrones anteriormente descritos han sido utilizados en;

Modelo-Vista-Controlador: toda la aplicación está basada en ella ,se proporcionan diferentes vistas pero el modelo sigue siendo el mismo.

Command : Paquete comandos. Se incluyen diferentes comandos que son fácilmente mantenibles y se autoreconocen.

Singleton: Se implementa en el reloj y solo permite una instancia de este.

Mediator: se implementa en la clase presentador.java para el patrón command.

# Discusión

Se trata de una práctica de gran complejidad, ya que dispone de numerosas clases, múltiples patrones implementados y una cantidad en horas de trabajo elevada.

# Conclusión

Esta práctica, nos ha servido para acercarnos más al lenguaje Java, a los conceptos de la POO, y a crear un buen código, ya que una vez más, hemos aplicado los factores de calidad del software.

Creemos que los resultados obtenidos de la solución implementada son positivos ya que, aunque no es la implementación deseada, cumple con la especificación de la práctica y nos sirve como solución provisional mientras ahondamos en el estudio de Threads que nos servirán para mejorar la implementación en las prácticas sucesivas.

Por lo dicho anteriormente, pensamos que hemos cumplido con los objetivos marcados.

# Líneas futuras

Además de la implementación de Dispatchers para la redirección de los comandos y la ejecución de varios ciclistas en una simulación de una carrera, en un futuro se añadirá una interfaz gráfica capaz de visualizar por pantalla de forma dinámica a cada uno de los ciclistas, su situación en el recorrido y los datos referentes a cada ciclista, cómo la velocidad o la distancia recorrida.

**Referencias bibliográficas utilizadas**

<http://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o>

# Anexos