

CAMPEONES

Oliver y Benji



Samuel Gómez Gutiérrez, Carlos Cortés Yagüe y Jesús Cobo Arrogante

1. Presentación del proyecto

En este proyecto se va a realizar el desarrollo de una aplicación de gestión de trabajo para un equipo de fútbol llamado New Team, que requerirá de funcionalidades básicas de creación, modificación y eliminación de registros así como administrar las entradas en una base de datos H2 en fichero y la gestión del almacenamiento local en el equipo leyendo y escribiendo en distintos formatos de archivo.

Toda la aplicación se gestionará a través de una interfaz construida sobre JavaFX, con un inicio de sesión que servirá para ejecutar la vista en modo usuario (sólo puede consultar los datos) o en modo administrador (puede editar los datos).

2. Presentación del equipo, reparto de tareas y análisis de riesgos

El proyecto fue realizado por [Samuel Gómez Gutiérrez](#), [Carlos Cortés Yagüe](#) y [Jesús Cobo Arrogante](#), tres estudiantes de Desarrollo de Aplicaciones Web en el IES Luis Vives de Leganés.

La gestión del proyecto ha sido la siguiente:

Se ha realizado el uso de una metodología de desarrollo en la que cada uno de los miembros del grupo se encargaría de implementar una funcionalidad o, varias interrelacionadas o, en su defecto, una parte de ellas llevando a cabo un desarrollo ágil y flexible ante modificaciones.

Por lo tanto, cuando alguno de los miembros ha sufrido un bloqueo con cualquier funcionalidad, ésta se ha conseguido sacar adelante con la ayuda y supervisión del resto de miembros del equipo.

Además, se ha considerado el riesgo de que alguno de los trabajadores no pudiese realizar las tareas asignadas a su parte del proyecto y en base a ello se ha destinado una partida de la estimación de costes del proyecto en concepto de horas extras para poder abarcar todos los requisitos de la aplicación.

3. Captura del gráfico de donde se muestre el modelo de flujo de trabajo GitFlow y se explique cómo se ha integrado y resuelto los conflictos

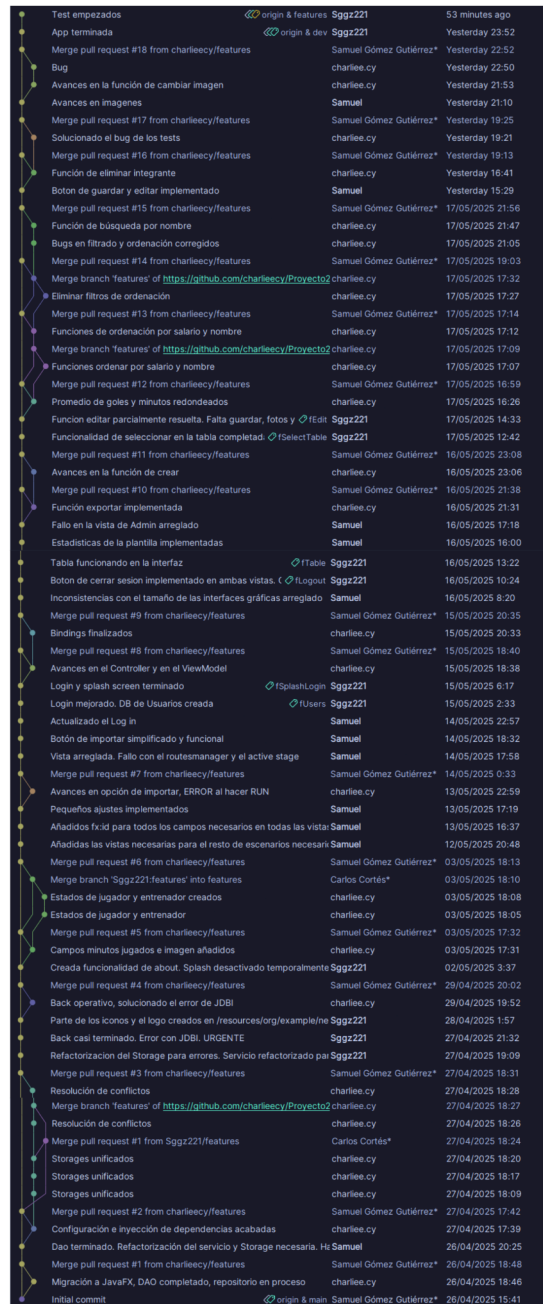


Gráfico de flujo de trabajo de git

Para el desarrollo de este proyecto se ha decidido trabajar mediante la metodología fork - pull request con GitHub. Samuel ha creado el repositorio principal con tres ramas:

- Main
- Dev
- Features

Por cada nueva funcionalidad a implementar, se ha abierto una rama en local derivada de features que, una vez terminada, se ha vuelto a fusionar con features y se ha subido a la rama Features del repositorio remoto.

Una vez que todas las funcionalidades de la aplicación se han completado, la rama Features del repositorio se ha fusionado con Dev y ésta, a su vez, con Main, quedando distribuido el flujo de trabajo como muestra la imagen.

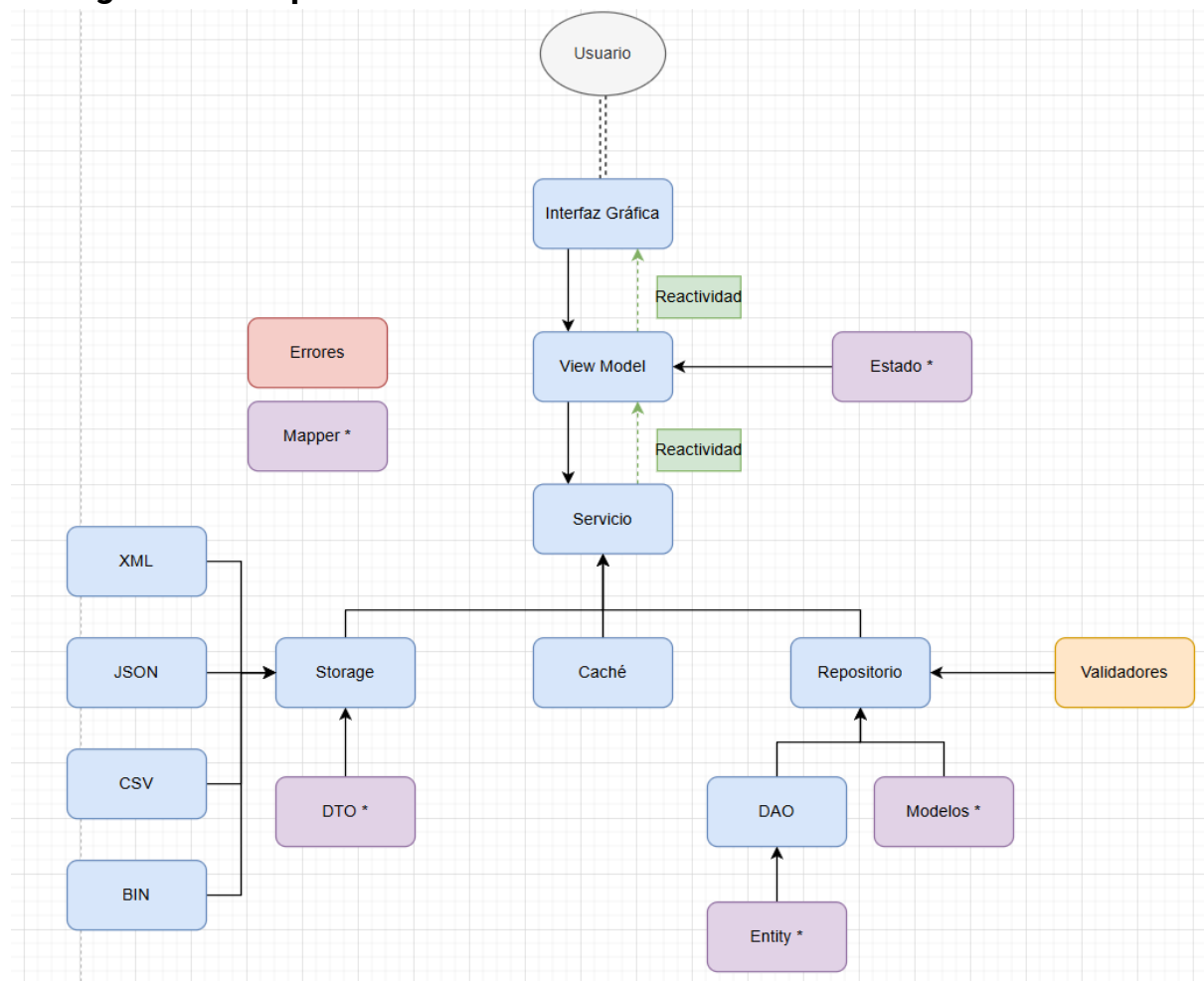
5. Especificación de requisitos funcionales, no funcionales y de información de forma razonada

- **Requisitos funcionales** (qué debe hacer la aplicación).
 - **RF1:** Realizar operaciones CRUD (Create, Read, Update y Delete) para gestionar los miembros del equipo.
 - **RF2:** Asignar un id único a cada miembro del equipo guardado.
 - **RF3:** Representar a los miembros del equipo como Jugadores o Entrenadores.
 - **RF4:** De todos los miembros, almacenar nombre, apellidos, fecha de nacimiento, fecha de incorporación, salario, país de origen y una imagen de perfil.
 - **RF5:** De los Jugadores, además, almacenar posición en el campo (portero, defensa, centrocampista o delantero), número de camiseta, altura, peso, número de goles anotados, partidos jugados Y minutos jugados.
 - **RF6:** De los Entrenadores, además, almacenar su área de especialización (entrenador de porteros, entrenador principal y entrenador asistente).
 - **RF7:** Validar los datos de los miembros antes de guardarlos para evitar errores o inconsistencias.
 - **RF8:** Permitir la importación y exportación de miembros desde y a ficheros CSV, JSON, XML y BIN a través de la interfaz de usuario siempre y cuando se esté *logueado* con una cuenta con privilegios de administrador.
 - **RF9:** Implementar una caché con filosofía LRU con un tamaño y caducidad de datos parametrizables a través de un fichero de configuración.

- **RF10:** Implementar una interfaz de usuario con iconos y un logotipo personalizados y únicos para la aplicación, asegurando su calidad y distinción del resto del mercado.
- **RF11:** Implementar una splash screen para mostrar mientras se abre la aplicación.
- **RF12:** Implementar un inicio de sesión con una base de datos de usuarios en la que se guarda el usuario y su contraseña usando la función de hash criptográfica de BCrypt.
- **RF13:** Crear dos vistas de la aplicación: usuario (sólo puede hacer visualizaciones de datos, filtrados y búsquedas) y admin (tiene permisos para crear, editar o eliminar integrantes del equipo).
- **RF14:** Conjunto de estadísticas en el *footer* de la aplicación que recogen datos como la media de goles, minutos jugados, y coste total de toda la plantilla (escogiendo los datos de entidades verdaderamente relevantes para no falsear la estadística final).
- **Requisitos no funcionales** (cómo debe funcionar la aplicación).
 - **RNF1:** Los datos de los miembros del equipo deben validarse para evitar errores e inconsistencias.
 - **RNF2:** La caché debe mejorar la velocidad de acceso a los datos.
 - **RNF3:** El menú debe ser intuitivo y facilitar el manejo de la aplicación a los usuarios.
 - **RNF4:** La aplicación debe implementar un sistema de logs para facilitar el seguimiento de la traza del programa y la corrección de errores.
 - **RNF5:** Todo el código debe estar documentado y el proyecto debe incluir la documentación generada por dokka.
 - **RNF6:** Todo el código debe ser testado.
 - **RNF7:** La aplicación debe poder ejecutarse desde la terminal gracias al archivo .jar
 - **RNF8:** Durante el desarrollo del proyecto, el equipo debe trabajar usando GitFlow/Git/Pull Request.
- **Requisitos de información** (peculiaridades sobre los tipos de datos que se gestionan).
 - **RI1:** Todos los miembros del equipo tendrán los siguientes campos comunes.
 - **Id:** De tipo Long para posibilitar una mayor escalabilidad del proyecto, al permitir un rango más amplio de valores que el tipo Int.

- **Nombre:** De tipo String. El nombre no puede estar vacío.
- **Apellidos:** De tipo String. Los apellidos no pueden estar vacíos.
- **Fecha de nacimiento:** de tipo LocalDate para que las fechas respeten el estándar ISO-8601. La fecha de nacimiento no puede ser posterior ni a la fecha actual ni a la fecha de incorporación.
- **Fecha de incorporación:** de tipo LocalDate para que las fechas respeten el estándar ISO-8601. La fecha de incorporación no puede ser posterior a la fecha actual ni anterior a la fecha de nacimiento.
- **Salario:** de tipo Double para poder representar los céntimos como cifras decimales. El salario no puede ser negativo.
- **País de origen:** de tipo String. El país de origen no puede estar vacío.
- **Imagen:** de tipo String. No puede estar vacía. Si no la hay, se selecciona una por defecto incluida en resources.
- **RI2:** Los miembros del equipo que sean Jugadores tendrán los siguientes campos específicos.
 - **Posición:** de tipo Posición (una enum class que contiene los valores portero, defensa, centrocampista y delantero).
 - **Dorsal:** De tipo Int, ya que solo se admitirán dorsales del 1 al 99.
 - **Altura:** De tipo Double, para poder representar los cms como cifras decimales. La altura no puede ser negativa ni superior a 3m.
 - **Peso:** De tipo Double, para poder representar los g como cifras decimales. El peso no puede ser negativo.
 - **Goles:** De tipo Int. Los goles no pueden ser negativos.
 - **Partidos jugados:** De tipo Int. Los partidos jugados no pueden ser negativos.
 - **Minutos jugados:** De tipo Int. Los minutos jugados no pueden ser negativos
- **RI3:** Los miembros del equipo que sean Entrenadores tendrán el siguiente campo específico.
 - **Especialidad:** de tipo Especialidad (una enum class que contiene los valores entrenador de porteros, entrenador principal o entrenador asistente).

6. Diagrama de arquitectura de software



Para visualizar el diagrama de clases, visualice [este video](#).

7. Herramientas tecnológicas usadas

- Entorno de desarrollo IntelliJ IDEA de JetBrains.
- Librería Logger para el uso de logs en la aplicación.
- Dokka para generar una documentación profesional.
- Git para el control de versiones y gestión del flujo de trabajo.
- GitHub para el trabajo en equipo.
- Librería Result para Railway Oriented Programming.
- Librería Caffeine para la caché.
- Librería BCrypt de jbcrypt.
- Librería open de Vaadin.

- Librería para la serialización de JSON.
- Librería para la serialización de XML.
- Librería JDBC para la gestión de bases de datos.

8. Justificación de elecciones de cada uno de los elementos software existente justificando de las opciones posibles y por qué se ha decantado el equipo por ella, justificado pros y contras de la solución parcial aportada por dicho elemento

No hubo muchos problemas a la hora de decidir por un diseño en concreto, pero cabe destacar:

- Que por consecuencia de la herencia en la que se basó el diseño principal de los modelos, se tuvo que decidir si los campos específicos para cada clase (tipo de entrenador, posición) se implementan con interfaces o con *enum class*.

Al final se decidió usar *enum class* debido a que no se iba a poder sacar tanto partido a las interfaces porque las clases no tenían comportamientos específicos dependiendo de su rol puesto que es un programa meramente de gestión. Las *enum class* por el contrario proveían de una implementación mucho más sencilla y a la hora de convertir los datos a los ficheros fue un proceso casi directo, cosa que con las interfaces no hubiera sido tan sencillo.

- La otra principal disyuntiva que se presentó fue a la hora de ordenar y filtrar los integrantes en la interfaz gráfica. Básicamente había dos opciones:
 - Generar una consulta SQL en el DAO para cada operación de filtrado y ordenación necesaria.
 - Aprovechar toda la lista de integrantes ya rescatada del servicio y filtrar sin necesidad de llamar de nuevo a la base de datos.

Finalmente se optó por la segunda opción, delegando la carga de ordenación y filtrado de datos en el lado del cliente para no saturar con peticiones a la base de datos.

- Crear una expresión regular altamente restrictiva para usuarios y contraseñas para acceder al programa para así evitar ataques por inyección de SQL, eliminando la posibilidad de uso de caracteres propensos a generar comentarios u operaciones en los dialectos SQL.
- Decidir entre:
 - Abrir una ventana modal para las acciones de editar y guardar un nuevo integrante.
 - Aprovechar la parte del detalle de la vista para usarla en la recogida de los datos a editar o guardar.

Finalmente se optó por la segunda opción debido a que, pese a que implicaba más complicaciones a la hora de afinar el controlador para modificar la apariencia y funcionalidades de sus distintos campos y botones en función de la acción seleccionada, a nuestro parecer resultaba más limpia, elegante e intuitiva a nivel de experiencia de usuario.

9. Principios SOLID aplicados y ejemplos de su implementación y uso en el código

Los principios SOLID usados en el proyecto son:

S (Single responsibility principle) Principio de responsabilidad única: Principio que dictamina que cada elemento del software debe realizar una única función y delegar el resto del trabajo a otras funciones. Ejemplo:

```
private fun disableAll() {  Sggz221
    disableComunes()
    disableJugador()
    disableEntrenador()
}
```

La función `disableAll()` no deshabilita todos los campos en la vista en un sólo bloque de código. Esto favorece el mantenimiento de código y escalabilidad a largo plazo a la vez que permite su reutilización en otras funcionalidades

O (Open/Close principle) Principio abierto/cerrado: Una clase debe estar cerrada a la su modificación pero abierta a su extensión. Ejemplo:

```

fun Entrenador.copy(  Sggz221
    newId: Long= this.id,
    newNombre: String= this.nombre,
    newApellidos: String = this.apellidos,
    new_fecha_nacimiento: LocalDate = this.fecha_nacimiento,
    new_fecha_incorporacion: LocalDate = this.fecha_incorporacion,
    newSalario: Double = this.salario,
    newPais: String = this.pais,
    newIsDeleted: Boolean = this.isDeleted,
    newEspecialidad: Especialidad = this.especialidad,
    timeStamp: LocalDateTime = LocalDateTime.now()
): Entrenador {
    return Entrenador(
        id = newId,
        nombre = newNombre,
        apellidos = newApellidos,
        fecha_nacimiento = new_fecha_nacimiento,
        fecha_incorporacion = new_fecha_incorporacion,
        salario = newSalario,
        pais = newPais,
        createdAt = timeStamp,
        updatedAt = timeStamp,
        isDeleted = newIsDeleted,
        especialidad = newEspecialidad,
    )
}

```

En este ejemplo se ve cómo en lugar de modificar la clase Entrenador, se extiende para añadir un comportamiento.

L (Liskov substitution principle) Principio de sustitución de Liskov: Los objetos de un programa pueden ser reemplazados por sus subtipos sin alterar su correcto funcionamiento. Ejemplo:

<ul style="list-style-type: none"> Entrenador Especialidad Integrante Jugador Posicion 	<pre> private val logger = logging() private val equipo = mutableMapOf<Long, Integrante>() private var nextId = 1L private var validator = IntegranteValidator() </pre>
---	---

En este ejemplo se muestra una herencia de clases, en la que la superclase **Integrante**, la cual es una clase abstracta y no se puede instanciar, está presente en todo el programa de manera que todas las colecciones de este son de Integrante. Esto es posible debido a que **Jugador** y **Entrenador** son subclases de Integrante y todos los comportamientos de Integrante lo heredan estas dos y donde se espera un Integrante se puede incluir o bien un Jugador o bien un Entrenador. Es una consecuencia del **polimorfismo**.

I (Interface segregation principle) Principio de segregación de interfaces: Es mejor tener varias interfaces con pocas funciones que una interfaz más grande que tenga muchas funciones que no todas las clases pueden usar y deben implementar igualmente. Ejemplo:

```

interface Service {  🐞 charliee.cy
    fun importFromFile(filePath: String)  🐞 charliee.cy
    fun exportToFile(filePath: String)  🐞 charliee.cy

    fun getAll(): List<Integrante>  🐞 charliee.cy
    fun getById(id: Long): Integrante  🐞 charliee.cy
    fun save(integrante: Integrante): Integrante  🐞 charliee.cy
    fun update(id: Long, integrante: Integrante): Integrante  🐞 charliee.cy
    fun delete(id: Long): Integrante  🐞 charliee.cy
    fun deleteLogical(id: Long, integrante: Integrante): Integrante  🐞 charliee.cy
}

```

En el ejemplo se observa como una interfaz Service tiene una serie de métodos que nosotros implementamos en la clase servicio. Si en un futuro la aplicación obtuviese una funcionalidad que requiriese un servicio distinto pero con más métodos, estos métodos deberían de ir en una interfaz a parte en lugar de aumentar el tamaño de ésta puesto que el actual servicio no haría uso de las nuevas funciones.

10. Patrones aportados en la solución y ejemplos de uso

Patrón MVC Model View Controller: Este patrón se basa en el uso de clases de manera que su responsabilidad se pueda diferenciar en las siguientes capas:

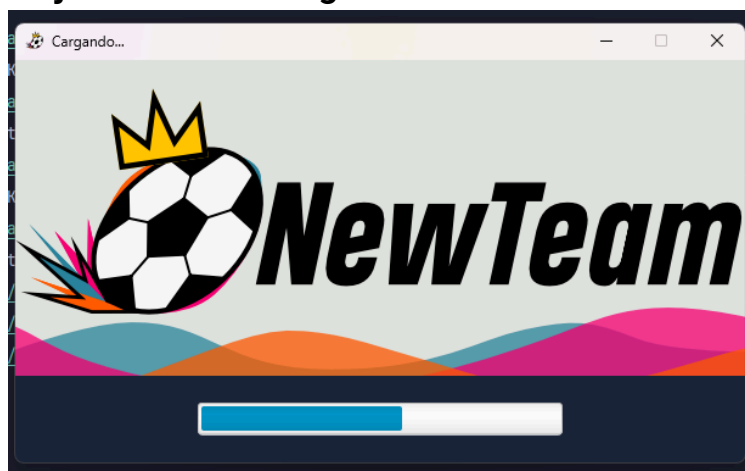
- Modelos: Son las abstracciones del programa y es lo que usan las clases del controlador para gestionar el programa. Aquí entrarían las clases Entrenador, Jugador, Integrante, las *enum class*, etc.
- Controlador: Es el encargado de gestionar todo el programa. Realización de cálculos, empleo de los modelos, gestión en memoria, etc.
- Vista: La información extraída del controlador es gestionada en la vista, la cual da un formato a la información, la muestra en pantalla y sirve como intermediario entre el usuario y el programa.

Patrón Singleton: Este patrón se basa en crear una única instancia de un objeto en tiempo de ejecución impidiendo múltiples instancias del mismo objeto.


Este patrón se cumple en la creación del objeto **Configuration**.

Model-View-ViewModel: Se usa en la parte de la interfaz gráfica de usuario para conseguir que los datos mostrados corresponden y reaccionen en tiempo real a los cambios en la parte del servidor y viceversa

11. Ejemplos de ejecución del código



Splash screen



Iniciar Sesión en NewTeam Manager

Usuario

Introducir texto....

Contraseña



Introducir texto...

Iniciar Sesión

Acerca de

Inicio de sesión

Requisitos funcionales 				
RF 	Nombre 	Horas estimadas 		
RF1	Operaciones CRUD	4		
RF2	Id único	0,5		
RF3	Modelos (jugador y entrenador)	2		
RF4	Campos comunes	1		
RF5	Campos específicos jugador	2		
RF6	Campos específicos entrenador	2		
RF7	Validador de datos	4		
RF8	Importación y exportación CSV, JSON, XML, BIN	12		
RF9	Cache LRU	5		
RF10	Interfaz de usuario	25		
RF11	Splash screen	4		
RF12	Inicio de sesión Bcrypt	2		
RF13	Vistas admin y usuario	10		
RF14	Estadísticas (goles, minutos, salarios)	3		
		76,5		

Requisitos no funcionales ▾ 		
RNF ▾	Nombre ▾	Horas estimadas ▾
RNF1	Validación de datos	2
RNF2	Rapidez de acceso mediante caché	4
RNF3	Facilidad de uso del menú	3
RNF4	Logs para debug	2
RNF5	Documentación del código	7
RNF7	.jar ejecutable	2
RNF8	Uso de GitFlow/Git/Pull Request	15
		35
Requisitos de información ▾ 		
RI ▾	Nombre ▾	Horas estimadas ▾
RI1	Campos comunes	5
RI2	Campos de jugador	6
RI3	Campos de entrenador	1
		12
HORAS DE TRABAJO TOTALES		
		123,5
PRECIO POR HORA DE TRABAJO		
		135,00 €
PRECIO POR HORA EXTRA DE TRABAJO		
		160,00 €
COSTE TOTAL DE LAS HORAS DE TRABAJO		
		16.672,50 €

ESTIMACIÓN DE COSTES				
Nº	Concepto	Desglose		Coste
		Precio por hora	Horas	
1	Horas de trabajo estimadas por requisitos	135,00 €	123,5	16.672,50 €
		Precio por hora extra	15% de las horas totales	
2	Horas extra para cubrir imprevistos	160,00 €	18,53	178,53 €
		Costes antes de margen	17% sobre los costes	
3	Margen de beneficios	16.851,03 €	0,17	2.864,67 €
				19.715,70 €