**Diseño y Arquitectura del Software**

**Grado Ingeniería de Software**



**Práctica 1**

****

|  |
| --- |
| **GRUPO 1**  **Carlos Alejandro A. (**[**c.alejandro.2019@alumnos.urjc.es**](mailto:c.alejandro.2019@alumnos.urjc.es)**)| Rodrigo Montilla F. | Javier Cai L. | Carlos García P. | Iván Gallego M. | Carlos Hermán A.** |

[**1. Roles**](#_dd7t5uhgy1bi)2

[**2. ADMentor**](#_peya9mpzcvy)2

[**3. Resultado Iteraciones**](#_rsxw4meec0jj)6

[**Semana 1: Tarea 1**](#_59m4ovf69f4s)6

[**Requisitos**](#_bz97m2ld998i)6

[**Semana 2: Iteración 1**](#_u5v4h7oewkjy)7

[**Estilo por Eventos**](#_76k8lx6p36q1)7

[**Communication Broker**](#_8u5sk2hqe0ra)9

[**Semana 3: Iteración 2**](#_39d83wrow6t7)11

[**Factory Method**](#_udox3s7zevxj)11

[**Abstract Factory**](#_emnv383k3g63)11

[**Observer**](#_pjwcbykfdw8n)12

[**Semana 4: Iteración 3**](#_zicvz0o88q2j)15

[**Facade**](#_9hxd4dbmftff)15

[**Strategy**](#_9vqkkdh4beq)16

[**Semana 5: Iteración 4**](#_uafzysq867w8)18

[**Chain of Responsibility**](#_r8nkk2ufj04m)18

[**Base de Datos MySQL**](#_7s8el67bj7lg)19

[**Base de Datos MariaDB**](#_bpiom7gwrkfd)19

[**4. Conclusiones en base a las decisiones**](#_px2pt37wghjo)22

[**5. Bibliografía**](#_snzhg6h8ldg6)22

[**6. Tiempos Estimados**](#_cpezd1881x8t)23

# **1. Roles**

**Arquitectos Software Senior (ASS):**

* Rodrigo Montilla Fernández
* Carlos Alejandro Álvarez

**Arquitectos Software Junior (ASJ):**

* Iván Gallego Morales
* Carlos Hernán Andrés

**Arquitectos Software Cognitivos (ASC):**

* Javier Cai Lin
* Carlos García Pérez

# **2. ADMentor**

ADMentor es una herramienta que da soporte a un lenguaje AD de modelado orientado a crear decisiones de arquitectura.

AD es un lenguaje de modelado diseñado para extraer el conocimiento de proyectos anteriores y utilizarlos como pautas de decisión en proyectos futuros. Un proyecto AD típico comienza con uno o más *espacios de problemas* existentes. Luego, los arquitectos derivan un *espacio de solución* concreto enriquecido con decisiones personalizadas específicas para el proyecto y documentan las opciones que se han elegido para resolver los problemas de diseño generales y específicos del proyecto. De manera que a partir de unos problemas de diseño se puede modelizar a través de ADMentor.

Esta herramienta ha sido ejecutada sobre Sparx Enterprise Architect, por lo que nos

hemos tenido que informar de cómo utilizar esta herramienta mediante tutoriales y

documentos facilitados por el profesor. El formado MADR nos ha permitido describir el

estado y la evolución de las decisiones. A continuación, mostraremos una serie de pasos

necesarios para el uso de esta herramienta:

* Primero, arriba a la izquierda hay un botón con el logo del software descargado donde haciendo ‘click’ se abrirá un panel con diferentes opciones, si seleccionamos “New Project” nos permitirá crear un proyecto nuevo. Además, este panel permite guardar, abrir, recargar o imprimir un proyecto dado.

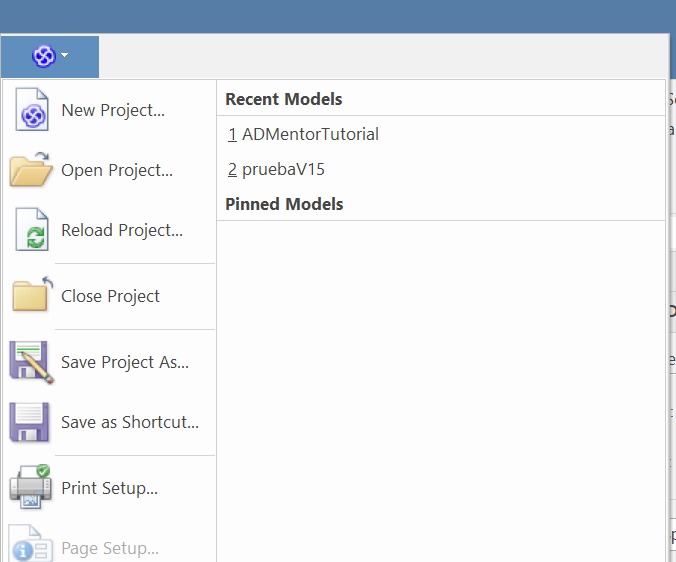


Ilustración . Crear un proyecto

* Segundo, cuando ya tengamos nuestro proyecto abierto, habrá que pinchar con el botón derecho del ratón donde pone “Model”, y seleccionar “Add a Model using Wizard”. Alternativamente, se puede pulsar la secuencia de teclas “Ctrl + Shift + m”.

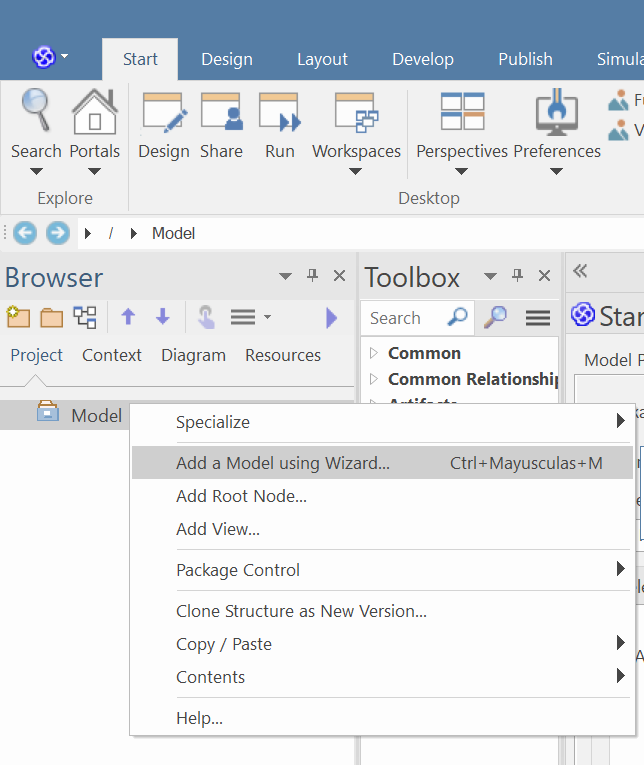


Ilustración . Crear un modelo

* En la ventana “Diagrama” podremos buscar, con el icono de lupa, “ADMentor”. Seleccionándolo, aparecerán 3 opciones. Nosotros nos centraremos en el *“Problem Space*” y el *“Solution Overview”*. Debemos asegurarnos de que la primera vez esté chequeada la casilla de “Crear paquete”.

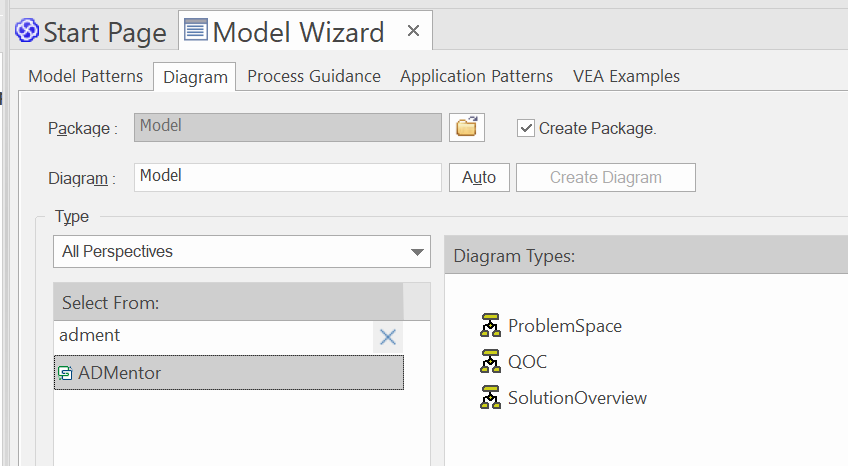


Ilustración . Crear el diagrama de ADMentor

* Tras haber creado el modelo, se podrá seleccionar en la pestaña de la izquierda, en el navegador. Al hacerlo, aparecerá la caja de herramientas donde se dispone de diferentes elementos, y el espacio del modelo, donde se podrán arrastrar desde la caja con el ratón para poder relacionarlos entre sí.

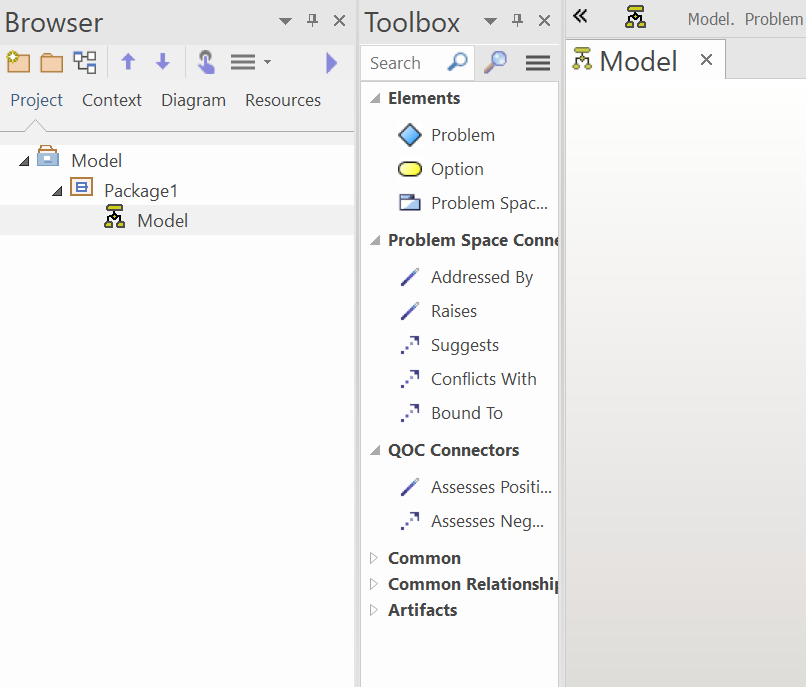


Ilustración . Vista de diagrama

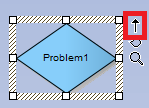
* Para acelerar el proceso, se dispone del método “Enlace rápido”, con el que, seleccionando la flecha que aparece al clickar en un elemento, y arrastrándola a un espacio libre, nos permite crear un nuevo elemento, con una relación más rápidamente.

Ilustración . Opción de enlace rápido

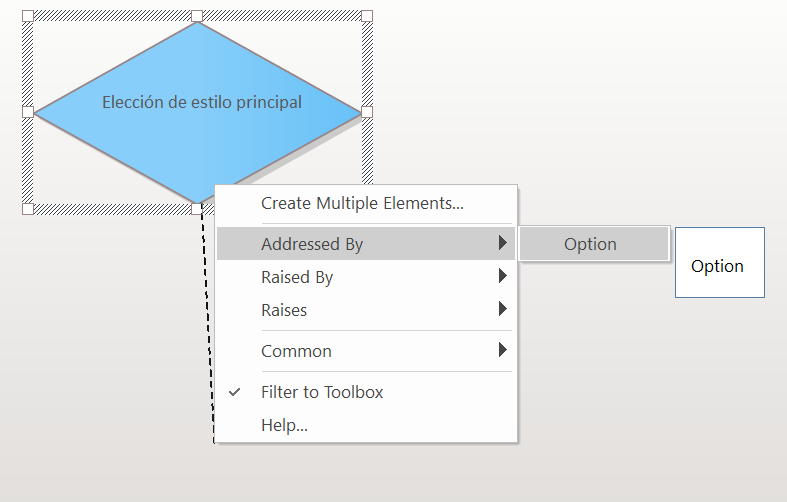


Ilustración . Crear opciones relacionadas con problemas

* También puede arrastrarse un nuevo elemento y relacionarlos entre ellos con las asociaciones de la caja de herramientas.

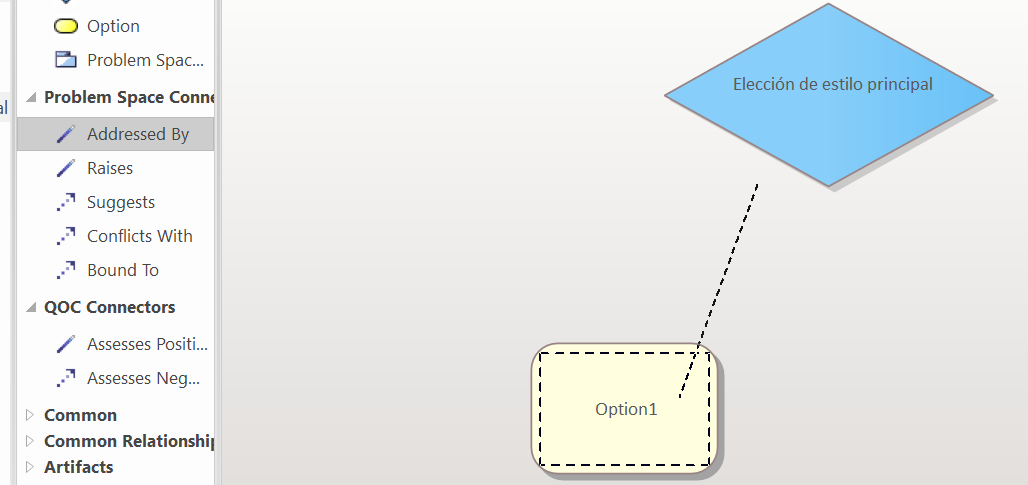


Ilustración . Crear una relación sin enlace rápido

* En nuestra práctica, lo utilizamos para presentar algunos problemas como la elección del estilo principal.

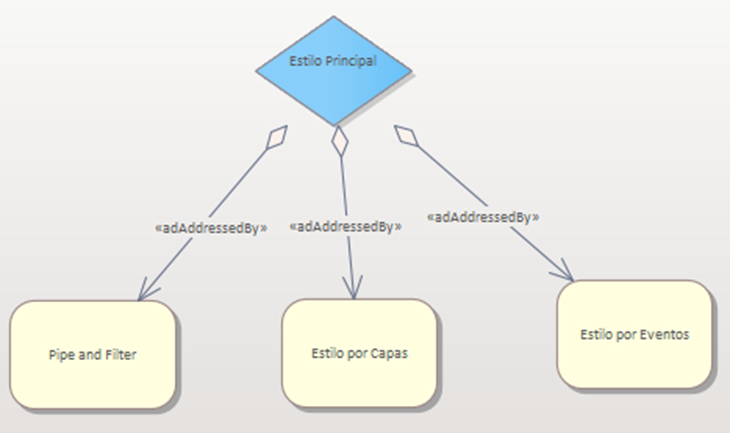


Ilustración . Espacio de problemas de la decisión principal

# **3. Resultado Iteraciones**

## **Semana 1: Tarea 1**

A continuación, se dispone la tabla con los requisitos funcionales que han sido extraídos del enunciado de la práctica en la Tarea 1: Semana 1:

|  |  |  |
| --- | --- | --- |
| **ID** | **Requisitos** | **Descripción** |
| RF-01 | Componente de visualización de datos | Un componente de visualización que muestre las analíticas en tiempo real del proceso productivo y las órdenes de trabajo. |
| RF-02 | Módulo de órdenes | Un módulo que asigne órdenes por operario y máquinas que van a fabricar cada componente. |
| RF-03 | Cockpit central | Cockpit donde se reciben los datos de los sensores y se visualizan las analíticas. Es un punto de entrada al software donde se gestionan todas las funcionalidades del software. |
| RF-04 | BBDD de inventario y órdenes | La BBDD deberá almacenar el estado del inventario y las órdenes de trabajo. |
| RF-05 | Múltiples familias de sensores | Existen tres familias de sensores, cada una de las cuales comparte cierta funcionalidad, pero dispone de otras diferentes entre una familia y otra. |
| RF-06 | Notificaciones de eventos | Este sistema enviará notificaciones a los usuarios suscritos a los diferentes eventos mediante un sistema de mensajería interno. |
| RF-07 | Componente de suscripción | Los usuarios podrán suscribirse a los diferentes eventos. |
| RF-08 | Soporte para familia de sensores comunicados entre sí | Existe una familia compuesta por tres sensores en los que el primero envía información al segundo y este al tercero que finalmente lo envía al centro de notificaciones. |
| RF-09 | Flujo de información sensores-cockpit | Existen más de 20 sensores IoT que recopilan datos sobre el estado de los dispositivos físicos de la factoría 4.0 y los envían al cockpit. |
| RF-10 | Decidir entre dos algoritmos | El software debe decidir en todo momento cuál de los 2 algoritmos es más adecuado utilizar. |
| RF-11 | Comunicación interna | Necesitamos un sistema que comunique a los sensores, el procesador y al centro de notificaciones (Cockpit), de manera que se comuniquen entre sí. |

Tabla 1. Requisitos extraídos

## **Semana 2: Iteración 1**

En esta segunda semana, empezamos a tomar las decisiones en función de los requisitos funcionales que se extrajeron la semana anterior. Primeramente, comenzamos por decidir qué tipo de estilo iba a ser el principal.

El estilo principal del sistema será un estilo por eventos porque, según como está escrito nuestro problema, es el que más cuadra. Llegamos a esta conclusión ya que hay un productor de eventos, en este caso serían los sensores, estos envían datos a un procesador de eventos, funcionando como gestor. Finalmente, el procesador envía órdenes al centro de notificaciones (Cockpit) que funciona como consumidor de eventos.

|  |  |
| --- | --- |
| ***Título de la decisión*** | Estilo por Eventos |
| ***ID*** | ADD-001 |
| ***Fecha*** | 03-11-2022 |
| ***Estado*** | Aceptada |
| ***Requisitos*** | RF-02: ”Módulo de órdenes”  RF-03: ”Cockpit central”  RF-09: ”Flujo de información sensores-cockpit” |
| ***Descripción*** | Está compuesta por productores y consumidores de eventos. Una vez que se detecta un evento, este se transmite a una plataforma donde se procesa de manera asíncrona para procesar los eventos. La plataforma de procesamiento ejecutará la respuesta adecuada para el evento y enviará la actividad a los consumidores correspondientes. |
| ***Decisiones alternativas*** | * **ADD-001.1 Estilo por Capas:** Consta en dividir la aplicación en capas, con la intención de que cada una tenga un rol muy definido * **ADD-001.2 Estilo Pipe and Filter:** Provee una estructura para los sistemas que procesan un flujo de datos. Cada paso de procesamiento se encapsula en un componente de filtro. Los datos pasan a través de tuberías entre filtros adyacentes. |
| ***Consecuencias positivas*** | * **Escalabilidad:** Es uno de los puntos más fuertes de esta arquitectura, pues permite que cada consumidor (Event Consumer) pueda escalar de forma independiente y reduce al máximo el acoplamiento entre los componentes. * **Despliegue:** Debido al bajo acoplamiento entre los componentes, es posible el despliegue sin preocuparse por dependencias o precondiciones, al final, los componentes solamente se suscriben para recibir eventos y reaccionar ante ellos. * **Performance:** Esta es una ventaja cuestionable, ya que al igual que en REST o SOA, EDA necesita pasar por una serie de pasos para completar una tarea, agregando retrasos en cada paso, desde colocar el Evento por parte del productor, esperar a que el consumidor lo tome, y generar nuevos Eventos, sin embargo, la naturaleza Asíncrona de EDA hace que esta desventaja se supere mediante el procesamiento en paralelo. * **Flexibilidad:** EDA permite responder rápidamente a un entorno cambiante, debido a que cada componente procesador de eventos tiene una sola responsabilidad y está completamente desacoplado de los demás, de esta forma, si ocurre un cambio, se puede aislar en un solo componente sin afectar al resto. Además, si aparece un nuevo requisito, solo es necesario regresar un nuevo tipo de procesador de eventos que escuche un determinado tipo de evento. |
| ***Consecuencias negativas*** | * **Testabilidad:** Una arquitectura distribuida y asíncrona agrega cierta complejidad a las pruebas, pues no es posible generar un evento y esperar un resultado para validar el resultado, en su lugar, es necesario crear pruebas más sofisticadas que creen los eventos y esperen la finalización del procesamiento del evento inicial más toda la serie de eventos que se podrían generar como consecuencia del evento inicial para validar el resultado final. * **Desarrollo**: Codificar soluciones asíncronas es complicado, pero aún más, es la necesidad de crear manejadores de errores más avanzados que permitan recuperarse en una arquitectura asíncrona, donde la falla en un procesador no significa la falla en el resto, lo que puede ocasionar la inconsistencia entre los sistemas. |
| ***Documentación*** | <https://learn.microsoft.com/es-es/azure/architecture/guide/architecture-styles/event-driven>  <https://reactiveprogramming.io/blog/es/estilos-arquitectonicos/eda> |

Tabla 2. ADD-001 Estilo por eventos

Continuando con la toma de decisiones, propusimos comenzar a desarrollar la decisión para la comunicación interna. Ya que necesitamos un sistema que comunique internamente los diferentes sistemas, como los sensores, el procesador y el centro de notificaciones (Cockpit) entre sí.

|  |  |
| --- | --- |
| ***Título de la decisión*** | Communication Broker |
| ***ID*** | ADD-002 |
| ***Fecha*** | 06-11-2022 |
| ***Estado*** | Aceptada |
| ***Requisitos*** | RF-11: ”Comunicación interna” |
| ***Descripción*** | Es un software que permite a las aplicaciones, sistemas y servicios comunicarse entre sí e intercambiar información. Es un mecanismo mediador de la comunicación entre aplicaciones, permitiendo minimizar el grado de conocimiento mutuo que estas aplicaciones necesitan tener, para poder intercambiar mensajes, implementando así efectivamente su desacoplamiento |
| ***Consecuencias positivas*** | * **Acoplamiento débil:** El cliente realiza una solicitud y no necesita conocer los otros servicios, por lo que no necesita usar un mecanismo de descubrimiento para encontrar la ubicación de las otras instancias de servicio. * **Comunicación más flexible** |
| ***Consecuencias negativas*** | * **Posible cuello de botella de rendimiento:** Podría ser un cuello de botella de rendimiento. * **Punto de fallo potencial:** Debe estar accesible continuamente. * **Complejidad operativa adicional:** Communication Broker es otro componente de un sistema que debe instalarse, configurarse y mantenerse. |
| ***Documentación*** | <https://betterprogramming.pub/why-do-we-need-message-broker-7382ce0e46c6> |

Tabla 3. ADD-002 Communication broker

Después de recibir las decisiones finales los Arquitectos Software Junior (ASJ) crearon el siguiente UML:

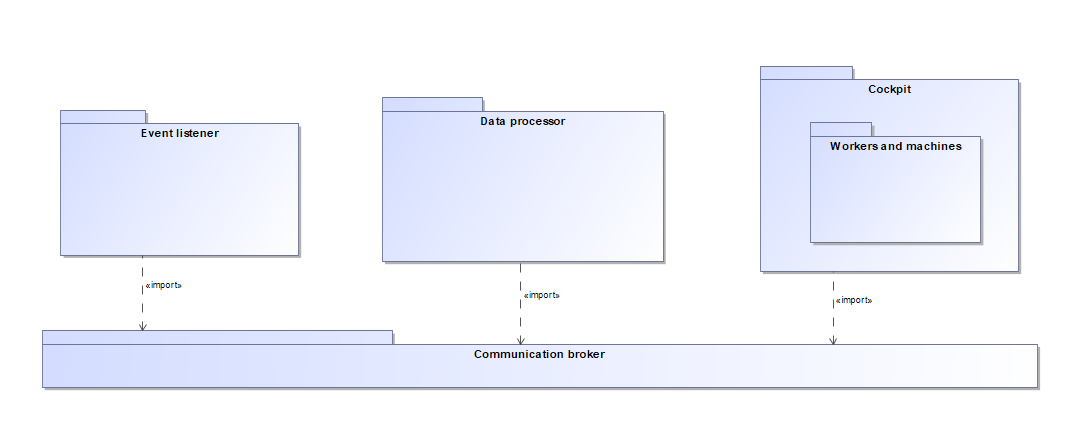


Fig 1. UML Iteración 1

Con las decisiones tomadas por los arquitectos Senior, decidimos estructurar la arquitectura como una distribuida por eventos. A la izquierda, conteniendo los sensores, el observador de eventos, que los genera. En el centro, el procesador de datos que unifica lo que le llegue de diferentes familias de sensores. A la derecha, el cockpit, que recibirá los datos del procesador y distribuirá las órdenes a los empleados suscritos, así como gestionar el inventario. Los 3 componentes usan un *broker* de comunicación, lo que le aporta un subestilo de capas al sistema.

## **Semana 3: Iteración 2**

En esta semana vamos a desarrollar los sensores. Según el enunciado se clasifican en tres familias, cada una de las cuales comparte cierta funcionalidad, pero existen ciertas diferencias entre una y otra. Para llevarlo a cabo, utilizaremos algún patrón creacional para poder diferenciar las diferentes familias de sensores.

|  |  |
| --- | --- |
| ***Título de la decisión*** | Factory Method |
| ***ID*** | ADD-003 |
| ***Fecha*** | 08-11-2022 |
| ***Estado*** | Rechazado |
| ***Requisitos*** | RF-05: “Múltiples familias de sensores” |
| ***Descripción*** | Es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán. |
| ***Decisiones alternativas*** | * **ADD-003.1 Abstract Factory** |
| ***Documentación*** | <https://refactoring.guru/es/design-patterns/factory-method> |

Tabla 4.ADD-003 Factory Method

Tras una discusión entre los Arquitectos Software Senior (ASS) y los Arquitectos Software Cognitivos (ASC), decidieron no usar la decisión principal y en cambio optaron por su alternativa. Ya que el enunciado habla de familia de sensores, eligieron aceptar la decisión de usar el patrón Abstract Factory en vez del patrón Factory Method.

|  |  |
| --- | --- |
| ***Título de la decisión*** | Abstract Factory |
| ***ID*** | ADD-003.1 |
| ***Fecha*** | 09-11-2022 |
| ***Estado*** | Aceptada |
| ***Requisitos*** | RF-05: “Múltiples familias de sensores” |
| ***Descripción*** | Es un patrón de diseño creacional que proporciona una interfaz para crear familias de objetos relacionados o dependientes entre ellos sin especificar una clase en concreto. |
| ***Consecuencias positivas*** | * **Compatibilidad:** Puedes tener la certeza de que los productos que obtienes de una fábrica son compatibles entre sí. * **Acoplamiento:** Evitas un acoplamiento fuerte entre productos concretos y el código cliente. * **Principio de responsabilidad única:** Puedes mover el código de creación de productos a un solo lugar, haciendo que el código sea más fácil de mantener. * **Principio de abierto/cerrado:** Puedes introducir nuevas variantes de productos sin descomponer el código cliente existente. |
| ***Consecuencias negativas*** | * **Desarrollo:** Puede ser que el código se complique más de lo que debería, ya que se introducen muchas nuevas interfaces y clases junto al patrón. |
| ***Documentación*** | <https://refactoring.guru/es/design-patterns/abstract-factory> |

Tabla 5. ADD-003.1 Abstract Factory

Esta semana finalizamos las decisiones con los sistemas de notificaciones y suscripciones. Estos sistemas claramente se parecen a algo que usaría el patrón de suscripción *Observer*, pero también nos encontramos con que se parecían al patrón *Publish-Subscribe*. Tras una discusión decidimos ir con el *Observer*, ya que principalmente el contexto del problema es un entorno más local.

|  |  |
| --- | --- |
| ***Título de la decisión*** | Observer |
| ***ID*** | ADD-004 |
| ***Fecha*** | 08-11-2022 |
| ***Estado*** | Aceptada |
| ***Requisitos*** | RF-06: “Notificaciones de eventos”  RF-07: ”Componente de suscripción” |
| ***Descripción*** | Es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando. |
| ***Decisiones alternativas*** | * **ADD-003.1 Publish-Subscribe:** Es un patrón de mensajería donde los mensajeros, llamados "publishers" no envían directamente los mensajes a los suscriptores, sino que categorizan los mensajes en clases sin saber nada acerca de los suscriptores. |
| ***Consecuencias positivas*** | * **Principio abierto/cerrado:** Puedes modificar las clases que reciben las notificaciones sin necesidad de modificar al notificador. * **Fácil adición de relaciones entre objetos** |
| ***Consecuencias negativas*** | * **Herencia:** El propio patrón te obliga a usar herencias al crear el notificador * **Complejidad:** En caso de tener un programa ya desarrollado y querer implementar un observer, la complejidad del software puede aumentar mucho. * **Fuga de memoria:** A la hora de añadir o quitar observadores, habría que hacerlo de manera explícita, lo que puede generar fugas de memoria. * **Aleatoriedad:** A la hora de notificar a los suscriptores, estas notificaciones llegarán en un orden aleatorio. |
| ***Documentación*** | <https://refactoring.guru/es/design-patterns/observer>  <https://www.geeksforgeeks.org/observer-pattern-set-1-introduction/> |

Tabla 6. ADD-004 Observer

Finalmente, después de recibir las decisiones finales los Arquitectos Software Junior (ASJ) modificaron el UML:

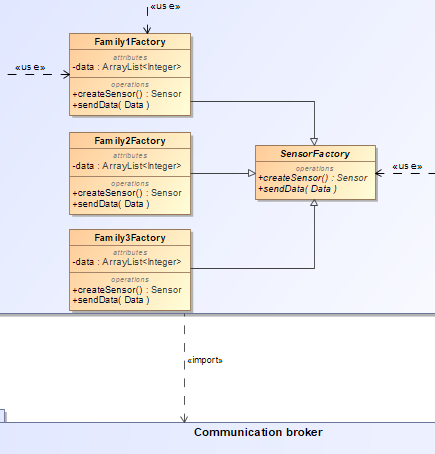


Fig 2. UML Iteración 2. Patrón Abstract Factory

Detalle del paquete ‘Data processor’, en el que se ha diseñado un patrón *Abstract Factory.* La fábrica de sensores contiene los métodos comunes a todos los sensores, pero es abstracta porque luego cada familia tiene sus particularidades. Se crean 3 dependencias de generalización, con fábricas concretas para crear los diferentes sensores.

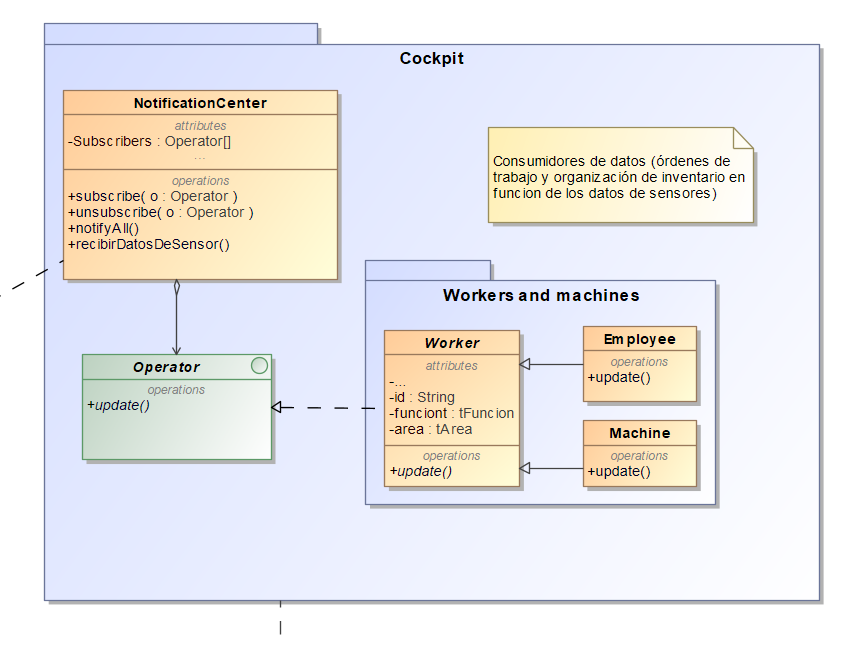


Fig 3. UML Iteración 2. Patrón Observer

Paquete íntegro *Cockpit* en el que se ha modelado un patrón *Observer.* El centro de notificaciones recibe los datos del procesador a través del *broker* de comunicación, y los distribuye a los suscriptores que tenga, divididos en humanos y máquinas, que los recibirán de diferente manera, adecuada a sus capacidades y características.

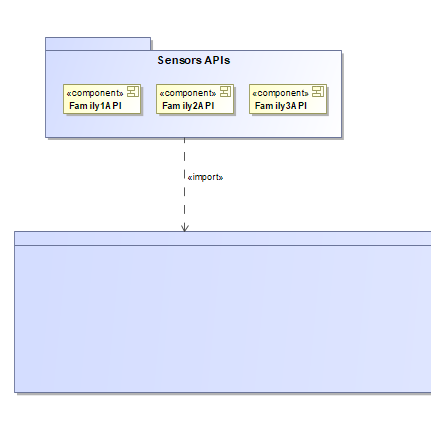


Fig 4. UML Iteración 2. Paquete de APIs

El paquete de sensores contiene las APIs para poder comunicarse con ellos y su hardware particular a la hora de recabar datos. El procesador de datos dispone del patrón *Abstract Factory* para poder lidiar con cada uno de los tipos.

## **Semana 4: Iteración 3**

En esta semana los Arquitectos Software Senior (ASS) se centraron en acabar todos los requisitos funcionales sobre la estructura interna del centro de notificaciones (Cockpit).

Empezando con el componente de visualización de datos, los Arquitectos Software Senior (ASS) y los Arquitectos Software Cognitivos (ASC) no encontraron durante su discusión ningúna alternativa que tuviese sentido poner.

|  |  |
| --- | --- |
| ***Título de la decisión*** | Facade |
| ***ID*** | ADD-005 |
| ***Fecha*** | 16-11-2022 |
| ***Estado*** | Aceptada |
| ***Requisitos*** | RF-01: “Componente de visualización de datos” |
| ***Descripción*** | Este patrón se utiliza para proporcionar una interfaz unificada de alto nivel para un conjunto de clases en un subsistema, lo que permite una mayor facilidad de uso. Simplifica el acceso a dicho conjunto de clases, ya que el cliente sólo se comunica con ellas a través de una única interfaz. |
| ***Consecuencias positivas*** | * Al separar al cliente de los componentes del subsistema, se reduce el número de objetos con los que el cliente trata, facilitando así el uso del subsistema. * Se promueve un acoplamiento débil entre el subsistema y sus clientes, eliminando o reduciendo las dependencias. * No existen obstáculos para que las aplicaciones usen las clases del subsistema que necesiten. De esta forma podemos elegir entre facilidad de uso y generalidad. * Puedes aislar tu código de la complejidad de un subsistema. |
| ***Consecuencias negativas*** | * Se tiene considerado el caso en que varios clientes necesiten acceder a subconjuntos diferentes de la funcionalidad que provee el sistema, podrían acabar usando sólo una pequeña parte de la fachada, por lo que sería conveniente utilizar varias fachadas más específicas en lugar de una única global. |
| ***Documentación*** | <https://refactoring.guru/es/design-patterns/facade>  <https://www.ecured.cu/Facade> |

Tabla 7. ADD-005 Patrón Facade

Acabando con el centro de notificaciones (Cockpit), encontramos que en el enunciado existen dos algoritmos y el software debe decidir en todo momento cuál de los 2 algoritmos es más adecuado utilizar. Para llevarlo a cabo, utilizaremos algún patrón de comportamiento para poder cambiar el comportamiento interno según el algoritmo que se elija.

|  |  |
| --- | --- |
| ***Título de la decisión*** | Strategy |
| ***ID*** | ADD-006 |
| ***Fecha*** | 16-11-2022 |
| ***Estado*** | Aceptada |
| ***Requisitos*** | RF-10: “Decidir entre dos algoritmos” |
| ***Descripción*** | Es un patrón de diseño de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables. |
| ***Decisiones alternativas*** | * **ADD-006.1 State:** Es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento de forma dinámica cuando su estado interno cambia. |
| ***Consecuencias positivas*** | * **Intercambiables:** Puedes intercambiar algoritmos usados dentro de un objeto durante el tiempo de ejecución. * **Aislables:** Puedes aislar los detalles de implementación de un algoritmo del código que lo utiliza. * **Sustituibles:** Puedes sustituir la herencia por composición. * **Principio de abierto/cerrado:** Puedes introducir nuevas estrategias sin tener que cambiar el contexto. * **Ahorro en memoria:** El cliente no va a necesitar todas las estrategias en todo momento, de modo que no necesitamos cargar un objeto con toda esa lógica dentro cada vez que queramos usarlo. * **Reutilización:** La misma estrategia puede usarse en diferentes contextos, evitando la duplicidad del código. |
| ***Consecuencias negativas*** | * Si solo tenemos un par de estrategias a implementar, puede ser un poco exagerado o excesivo implementar este patrón, ya que complicamos en exceso la casuística. * El cliente debe de conocer las diferencias entre las estrategias. |
| ***Documentación*** | <https://refactoring.guru/es/design-patterns/strategy>  <https://medium.com/all-you-need-is-clean-code/patr%C3%B3n-estrategia-strategy-pattern-654c6e9d2abe> |

Tabla 8. ADD-006 Patrón Strategy

Tras capturar las decisiones los Arquitectos Software Junior (ASJ) modificaron el UML:

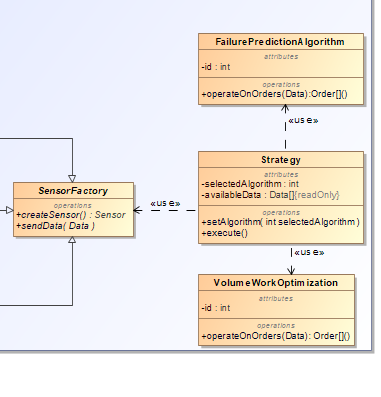


Fig 5. UML Iteración 3. Patrón Strategy

En el paquete del procesador, se aplicó un patrón *Strategy* sobre la fábrica abstracta de sensores, para que los datos que reciba de cualquier tipo de sensor pueda ser enviado al algoritmo y ejecutado. La clase *Strategy* puede elegir entre cada uno de los dos algoritmos disponibles, que reciben datos y devuelven órdenes a trabajadores.

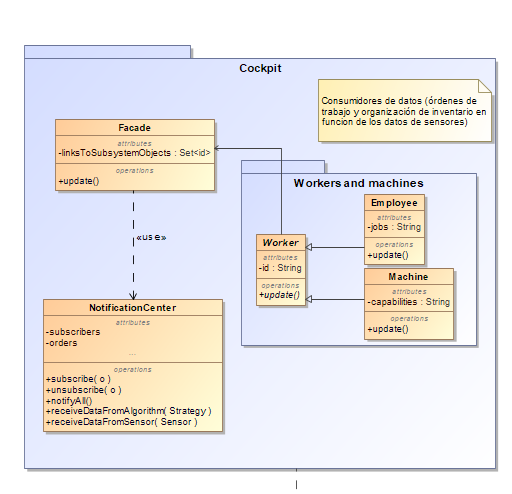


Fig 6. UML Iteración 3. Patrón Facade

Respecto al paquete del *Cockpit*, con el patrón *Observer* anteriormente incluido, se decidió interponer un patrón *Facade* para facilitar la comunicación del trabajador con la gran variedad de datos disponibles que llegan al centro de notificación, y que solo interactúe con aquellos que necesite.

## **Semana 5: Iteración 4**

Al ser la última semana, se llevaron a cabo las decisiones restantes, de manera que se pudo conseguir una visión más global del proyecto en sí y de así pudiendo confirmar que ciertas decisiones que hicimos en semanas anteriores eran correctas.

Esta semana la empezamos acabando con los sensores. En el enunciado habla de que existe una familia la cual está compuesta por tres sensores en los que el primero envía información al segundo y este al tercero que finalmente lo envía al centro de notificaciones.

Para este problema tanto los Arquitectos Software Senior (ASS) como los Arquitectos Software Cognitivos (ASC) tuvieron muy clara la solución, ya que durante las discusiones no pudieron encontrar ninguna alternativa que tuviera sentido.

|  |  |
| --- | --- |
| ***Título de la decisión*** | Chain of Responsibility |
| ***ID*** | ADD-007 |
| ***Fecha*** | 20-11-2022 |
| ***Estado*** | Aceptada |
| ***Requisitos*** | RF-08: “Soporte para familia de sensores comunicados entre sí” |
| ***Descripción*** | Es un patrón de diseño de comportamiento que te permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena. Básicamente, este patrón ayuda a encapsular acciones secuenciales sobre un objeto. |
| ***Consecuencias positivas*** | * **Controlable:** Puedes controlar el orden de control de solicitudes. * **Principio de responsabilidad única:** Puedes desacoplar las clases que invoquen operaciones de las que realicen operaciones. * **Principio de abierto/cerrado:** Puedes introducir nuevos manejadores en la aplicación sin descomponer el código cliente existente. |
| ***Consecuencias negativas*** | * Algunas solicitudes pueden acabar sin ser gestionadas. |
| ***Documentación*** | <https://refactoring.guru/es/design-patterns/chain-of-responsibility> |

Tabla 9. ADD-007 Patrón Chain of Responsibility

Acabando con los requisitos funcionales encontramos que existe una BBDD y esta deberá almacenar el estado del inventario y las órdenes de trabajo. Por tanto, la decisión consiste en qué tipo de BBDD es más adecuado para este problema.

|  |  |
| --- | --- |
| ***Título de la decisión*** | Base de Datos MySQL |
| ***ID*** | ADD-008 |
| ***Fecha*** | 20-11-2022 |
| ***Estado*** | Rechazada |
| ***Requisitos*** | RF-04: “BBDD de inventario y órdenes” |
| ***Descripción*** | Es un sistema de gestión de bases de datos relacionales de código abierto con un modelo cliente-servidor. Esto quiere decir que la información se encuentra centralizada en un servidor (normalmente tu hosting) que recibe y procesa las peticiones. Estas peticiones son realizadas por los diferentes clientes (usuarios que realizan consultas a la base de datos a través de herramientas como phpMyAdmin, por ejemplo) en forma de sentencias SQL. |
| ***Decisiones alternativas*** | * **ADD-008.1 MariaDB** * **ADD-008.2 Oracle Database:** Es un sistema de gestión de bases de datos relacionales de Oracle, el fabricante estadounidense de software y hardware. Como software de bases de datos, Oracle Database optimiza la gestión y seguridad de los conjuntos de datos creando esquemas estructurados a los que solo pueden acceder administradores autorizados. |
| ***Documentación*** | <https://www.hostinger.es/tutoriales/que-es-mysql> |

Tabla 10. ADD-008 Base de datos MySQL

Al no ser esta una decisión sobre un patrón o estilo los Arquitectos Software Senior (ASS) tuvieron que buscar mucha más información sobre las diferentes bases de datos y sus diferencias, ya que no sabían mucho al respecto.

Tal es el caso que durante la discusión entre los Arquitectos Software Senior (ASS) y los Arquitectos Software Cognitivos (ASC) decidieron no usar la decisión principal y en cambio optaron por su alternativa. Ya que vieron que las consecuencias positivas de MariaDB se adecuaban más al problema.

|  |  |
| --- | --- |
| ***Título de la decisión*** | Base de Datos MariaDB |
| ***ID*** | ADD-008.1 |
| ***Fecha*** | 22-11-2022 |
| ***Estado*** | Aceptada |
| ***Requisitos*** | RF-04: “BBDD de inventario y órdenes” |
| ***Descripción*** | MariaDB es un sistema de gestión de bases de datos derivado de MySQL con licencia GPL (General Public License). Introduce dos motores de almacenamiento nuevos. Tiene una alta compatibilidad con MySQL ya que posee las mismas órdenes, interfaces, API y bibliotecas, siendo su objetivo poder cambiar un servidor por otro directamente. |
| ***Consecuencias positivas*** | * **Retrocompatibilidad:** Es capaz de compatibilizar con versiones antiguas con facilidad. * **Motores de almacenamiento:** Es capaz de soportar mayor número de motores de almacenamiento que otras bases de datos SQL. |
| ***Consecuencias negativas*** | * **JSON**: Solo acepta tipos de datos JSON, no tiene compatibilidad con ningún otro * **Funcionalidades:** Varias funcionalidades de otras bases de datos SQL no son soportadas por MariaDB |
| ***Documentación*** | <https://www.hostingplus.com.es/blog/que-es-mariadb-y-cuales-son-sus-caracteristicas/>  <https://www.geeksforgeeks.org/difference-between-mysql-and-mariadb/> |

Tabla 11. ADD-008.1 Base de datos MariaDB

Tras las diferentes decisiones de diseños tomadas a lo largo de las iteraciones para la resolución de este ejercicio y las decisiones de esta semana, los Arquitectos Software Junior (ASJ) modificaron el UML de la semana pasada de la siguiente manera:

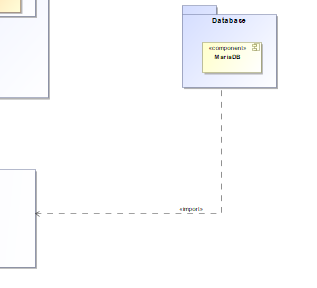


Fig 7. UML Iteración 4. Diseño de la base de datos

La base de datos, con un componente de API para poder ser usada, se comunica a través del *broker* de comunicación con el resto de los subsistemas.

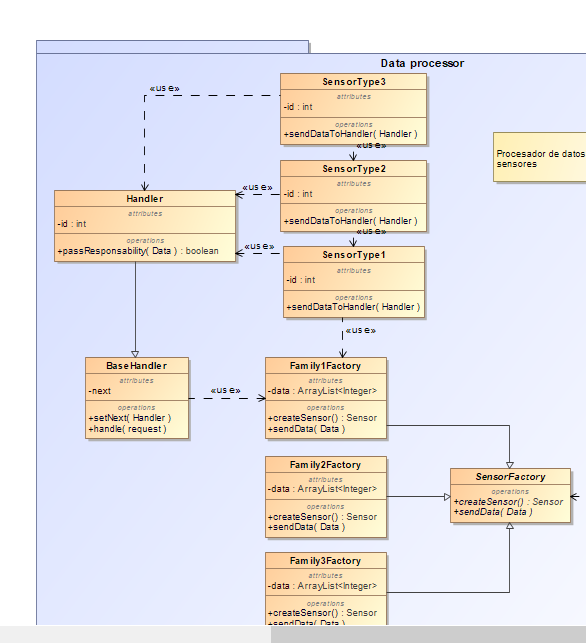


Fig 8. UML Iteración 4. Patrón Chain of Responsibility

Se seleccionó arbitrariamente la familia 1 para los sensores secuenciales. El patrón *Chain of Responsability* ha quedado intercalado entre cada uno de los sensores de dicha familia, de manera que antes de comunicarse con el siguiente sensor, envían sus datos al manejador que se encarga de verificar si los datos deben ser pasados por la cadena o no.

En conjunto, el diagrama tiene esta estructura:

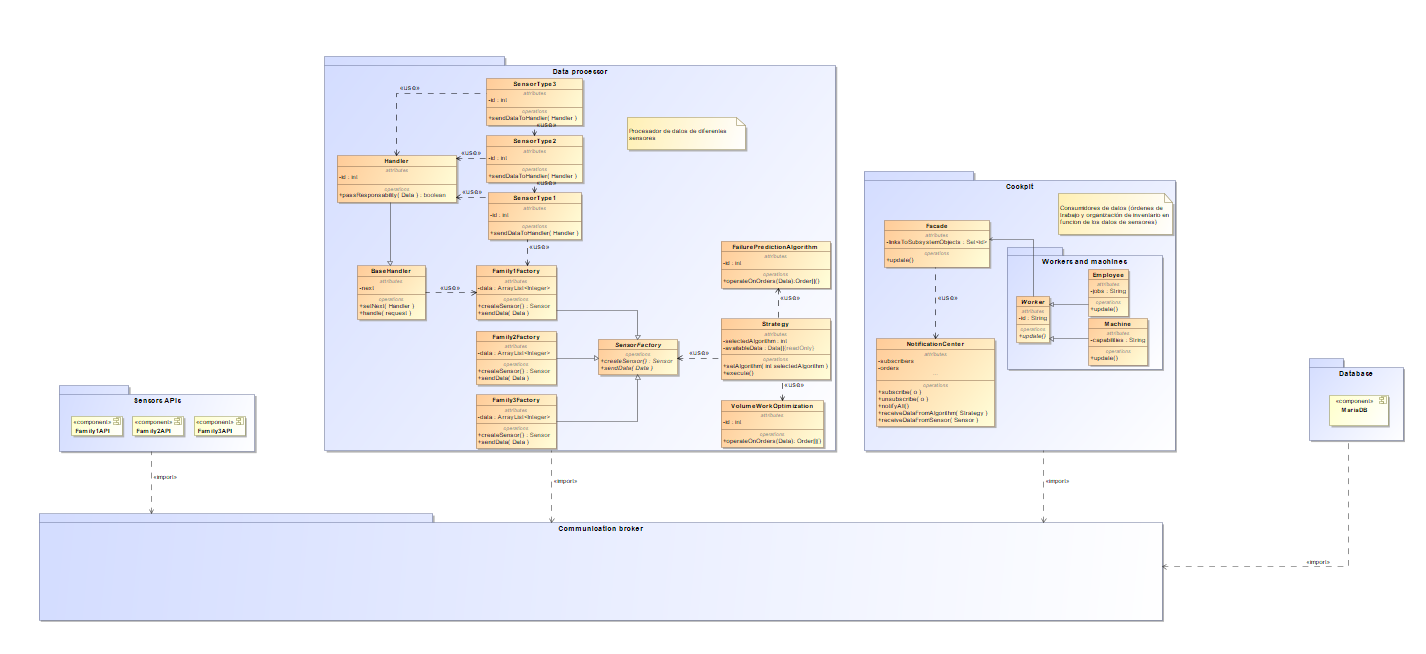


Fig 9. UML Iteración 4. Diseño final

Mantiene la arquitectura elegida en la primera iteración, habiendo diseñado el interior de los paquetes para permitir un funcionamiento acorde a lo descrito en el enunciado del problema.

# **4. Conclusiones en base a las decisiones**

La experiencia que hemos obtenido del trabajo realizado nos ha aportado muchas cosas. La primera y muy importante es la importancia de analizar múltiples veces un problema antes de decidirnos por una solución, ya que las constantes revisiones ofrecen diferentes puntos de vista que nos ayudan a observar la imagen general, con los pros y los contras de cada punto. También, la importancia de tener a alguien externo de la decisión inicial para poder discutir, en este caso los Arquitectos Software Cognitivos (ASC), que nos han ayudado a decidir entre varias decisiones y sus alternativas cuando no lo teníamos claro, además de explicar el motivo por el que en algunas habíamos decidido mal.

Consideramos crucial el uso de iteraciones pues nos permiten detectar errores en etapas tempranas de diseño y corregirlos, sea sustituyéndolos por alternativas mejores o alterando aquellos puntos que hacen fallar la decisión actual.

Finalmente el desarrollo de la práctica nos ha servido no solo para poner en práctica los conocimientos teóricos aprendidos en la asignatura, sino también para poder usar todo lo que hemos aprendido en el futuro, además de poder saber cómo funcionan los roles y las dinámicas del grupo en un sentido más práctico, puesto que al principio sí que es verdad que como todo el mundo era nuevo en este sistema de roles y más o menos nuevos en cuanto al conocimiento práctico de algunas cosas, ya que las habíamos visto únicamente teóricamente. Pero a medida que fuimos tomando decisiones, discusiones y arquitecturas fuimos mejorando en nuestro rol individual y la dinámica de nuestro grupo.

# **5. Bibliografía**

* **Estilo por Eventos, recuperado de:** <https://learn.microsoft.com/es-es/azure/architecture/guide/architecture-styles/event-driven>

<https://reactiveprogramming.io/blog/es/estilos-arquitectonicos/eda>

* **Qué es y como funciona el Communication Broker:** <https://betterprogramming.pub/why-do-we-need-message-broker-7382ce0e46c6>
* **El patrón Factory Method, recuperdado de:** <https://refactoring.guru/es/design-patterns/factory-method>
* **El patrón Abstract Factory, recuperado de:** <https://refactoring.guru/es/design-patterns/abstract-factory>
* **El patrón Observer, recuperado de:**

<https://refactoring.guru/es/design-patterns/observer>

<https://www.geeksforgeeks.org/observer-pattern-set-1-introduction/>

* **El patrón Facade, recuperado de:**

<https://refactoring.guru/es/design-patterns/facade>

<https://www.ecured.cu/Facade>

* **El patrón Strategy, recuperado de:**

<https://refactoring.guru/es/design-patterns/strategy>

<https://medium.com/all-you-need-is-clean-code/patr%C3%B3n-estrategia-strategy-pattern-654c6e9d2abe>

* **El patrón Chain of Responsibility, recuperado de:** <https://refactoring.guru/es/design-patterns/chain-of-responsibility>
* **Base de datos MySQL:**

<https://www.hostinger.es/tutoriales/que-es-mysql>

* **Base de datos MariaDB y diferencias con MySQL:**

<https://www.hostingplus.com.es/blog/que-es-mariadb-y-cuales-son-sus-caracteristicas/>

<https://www.geeksforgeeks.org/difference-between-mysql-and-mariadb/>

# **6. Tiempos Estimados**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Week** | **Iteration** | **Time in ADD (ASS)** | **Reflection Time (ASS-ASC)** | **Time in refined ADD (ASS)** | **Design ADD Time (ASJ)** |
| **1** | **1**  **(Requisitos)** | **158** | **52** | **61** | **35** |
| **2** | **1.1** | **116** | **38** | **24** | **15** |
| **3** | **2** | **94** | **43** | **33** | **97** |
| **4** | **3** | **124** | **61** | **26** | **70** |
| **5** | **4** | **105** | **57** | **38** | **63** |

Tabla 12. Tabla de tiempos